Introduction, and Outline to the field of Algorithmics.

Table of Contents

Introduction 5

Discussion 5

Runtimes 6

O-notation (asymptotic notation) 7

Exercise 8

Divide and conquer. 8

Simple binary search 9

The logarithmic function 10

Logarithmic time 11

Exercises 12

Polynomial Time 15
Decision Problems 17
Question 17
Other examples: The Travelling Salesperson Problem 17
Another example 19
Witnesses and verification, P, NP and Co-NP 20
Re-cap 21
Solution and Verification 22
Case Study: Shortest Paths 23
Shortest path decision 24
Subset Sum 27
Tautology 27
Food for thought 28
NP-Completeness and NP-Hardness 28
NP-Completeness 28
Satisfiability (SAT) 30
Consequences of NP-Completeness 31
Satisfiability $\geq_P$ 3-SAT 32
Satisfiability $\geq_P$ 3-SAT 33
3-SAT $\geq_P$ Independent Set 34
An example follows... 35
3-SAT $\geq_P$ Vertex Cover 35
NP-Complete Problems 37
Proof techniques 38
Practice 38
Moving forward - Optimisation Problems 39
Polynomial Time Algorithms 40
Divide and Conquer 42
Binary Search 43
Mergesort 43
Quicksort 43
Reduction (parallel programming pattern) 44
Greedy Algorithms 44
A simple example is finding the minimum number of coins to give as change. 45

Or Prim's algorithm for the minimum spanning tree problem 45
Even better, the algorithm for generating Huffman Codes is greedy 46
Dynamic Programming 48
Typesetting 49
Shortest paths 51
Linear and Integer Programming 56
Linear programming 56
A simple example 57
A little geometry and the simplex algorithm 57
Simplex algorithm 59
Integer programming 62
What does this mean? 63
Integer programming 64
Solution techniques 65
Heuristics 66
An example 67
Greedy heuristics 68
Try both heuristics on these 69
Generic algorithms 70
Practice 71
Escaping Local Optima 72
Local search 72
A generic local search process 74
Iterative improvement 75
Local and Global optimality 76
Metaphors galore 77
Escaping local optima 78
Simulated Annealing (SA) 80
The SA algorithm 80
Tabu Search 82
Tabu Search (TS) 83
Metaheuristics 84
Construction metaheuristics 84
GRASP 86

Task 87
Ant Colony Optimisation (ACO) 88
ACO for the TSP 90
Population vs Trajectory based techniques 91
Genetic Algorithms and Evolutionary Computing 92
GAs 93
Ridiculous metaphor 93
Hyperheuristics 94
PSPACE 100
Memetic Algorithms 101
Post's Correspondence Problem 101
Polyomino Tiling's 101

Algorithmics

Introduction

Algorithmics, also known as the study of algorithms, is a branch of computer science that deals with the design, analysis, and implementation of algorithms. Algorithms are step-by-step procedures used to solve problems and perform tasks, and are a fundamental part of computer science and many other fields.

The field of algorithmics is broad and encompasses many subfields, including:

- Algorithm design: This subfield focuses on the process of creating new algorithms, whether through formal mathematical methods or through more heuristic approaches.
- Algorithm analysis: This subfield deals with determining the time and space complexity of algorithms, which are measures of how well an algorithm scales as the size of the input increases. This includes the use of asymptotic notation, such as O(), Ω(), and Θ(), to describe the upper and lower bounds of an algorithm's performance.
- Algorithm engineering: This subfield is concerned with the practical aspects of algorithm design and analysis, including the implementation and testing of algorithms, as well as the design of efficient data structures.
- Computational complexity theory: This subfield deals with the study of the inherent difficulty of computational problems and the limits of computation. It includes the study of NP-completeness and the P vs NP problem.
- Randomized algorithms: This subfield deals with the design and analysis of algorithms that use randomness as a key element in their operation.
- Approximation algorithms: This subfield deals with the design and analysis of algorithms that are allowed to return approximate solutions to problems, rather than exact solutions.
- Online algorithms: This subfield deals with the design and analysis of algorithms that must make decisions based on input that is presented in a serial fashion, rather than all at once.
- Quantum algorithms: This subfield deals with the design and analysis of algorithms that take advantage of the unique properties of quantum mechanics to solve problems more efficiently than classical algorithms.

Algorithmics is a constantly evolving field, with new techniques and approaches being developed all the time. The development of new algorithms is driven by the need to solve increasingly complex problems, as well as the desire to improve the efficiency and scalability of existing algorithms. The field of algorithmics is also heavily interdisciplinary, with links to other fields such as mathematics, operations research, artificial intelligence, and computer engineering.

Discussion

How hard is it to solve the following three problems:

1. Given three integers a,b and c - find out if $a + b = c$.
2. Given a set of integers S = {a,b,c….} and an integer k, find out if all elements of S add up to k
3. Given a set of integers S = {a,b,c….} and an integer k, find out if any subset of S adds up to k

Can you think of a way of solving each problem?

Are some solutions likely to be faster than others? How do we define ‘fast’?

The first problem is very simple to solve and can be done in O(1) time, as it only requires you to check if a+b equals c.

The second problem can also be easily solved in O(n) time, where n is the number of elements in the set S. You can simply iterate through the elements of the set and add them up, then check if the sum equals k.

The third problem is more challenging and is known as the "subset sum problem." One way to solve it is to use a brute-force approach, where you enumerate all possible subsets of S and check if any of them add up to k. This solution would have a time complexity of O(2^n), where n is the number of elements in the set S.

There may be faster solutions to the third problem depending on the specific constraints of the problem and the nature of the data. For example, if the elements of the set S are all positive, you can use a dynamic programming approach to solve the problem in O(n*k) time, where k is the target sum.

In general, we define an algorithm to be "fast" if it can solve a problem in a reasonable amount of time, given the size of the input data. The time complexity of an algorithm gives us a way to measure how the running time of the algorithm scales with the size of the input data. An algorithm with a lower time complexity will generally be faster than one with a higher time complexity, for large inputs.

Runtimes

The runtime of our program depends on two things:

- The number of data items to be processed
    - In the previous example, the more items there are, the longer the program loops
    - Runtimes must be expressed as a function of the input size (usually denoted by n)
- The values of the data items themselves
    - Note that some sections of code only execute when data items meet certain conditions
    - Also, we may terminate early if we’ve found the answer without processing all the data.
    - Consequently, we tend to analyse the *worst-case* runtime for any n data items.

So the worst case runtime of the program on the previous page might be something like $34 + 12n + 23n^2$ (measured in clock cycles).

The runtime of an algorithm is the amount of time it takes for the algorithm to complete its task, given a specific input size. The time complexity of an algorithm is a measure of how the runtime of the algorithm scales with the size of the input.

For example, consider an algorithm that sorts a list of numbers. The time complexity of the algorithm tells us how the running time of the algorithm increases as the size of the input list grows. If the time complexity of the algorithm is O(n), this means that the runtime of the algorithm increases linearly with the size of the input. For example, if the input size doubles, the runtime of the algorithm will also approximately double.

In general, we want algorithms to have low time complexity, so that they can solve problems efficiently even for large inputs. Algorithms with higher time complexity may take too long to complete for very large inputs, making them impractical to use in many situations.

O-notation (asymptotic notation)

It should be clear that trying to express runtimes in terms of an exact number of clock cycles is unwieldy:

- It gives us some potentially very nasty formulae for runtimes
- It's architecture dependent (more importantly)

So instead, we try to find a shorthand measure of runtime, whereby we place an *upper bound* on the runtime as the input size grows very large (or *tends to infinity* to use the mathematical expressions)

For example, considering $34 + 12n + 23n^2$, the first two terms become negligibly small compared to $23n^2$ as n becomes very large.

O-notation, also known as asymptotic notation, is a way of expressing the time complexity or space complexity of an algorithm. It gives us a rough idea of how the runtime or memory usage of an algorithm scales with the size of the input.

O-notation is usually expressed using big O notation, which gives an upper bound on the time or space complexity of an algorithm. For example, if an algorithm has a time complexity of O(n), this means that the runtime of the algorithm grows at most linearly with the size of the input. This is a rough estimate, and the actual runtime of the algorithm may be faster.

There are several other variations of O-notation, including omega notation (Ω) and theta notation (Θ), which give lower and exact bounds on the time or space complexity of an algorithm, respectively.

O-notation is useful because it allows us to compare the time or space complexity of different algorithms, even if they have different constants. For example, an algorithm with a time complexity of O(n^2) will generally be slower than an algorithm with a time complexity of O(n), even if the constants in the O-notation expressions are different.

It is important to note that O-notation only gives a rough estimate of the time or space complexity of an algorithm, and the actual runtime or memory usage of an algorithm may differ from the O-notation estimate.

To enable a more effective and concise way to compare and discuss runtimes, we use the following intuition.

- A function $f(n)$ is said to be *order of* $g(n)$ or $O(g(n))$ if for all values of n greater than c, $f(n) < k * g(n)$
    - c and k are constants
- So given our previous example, the worst case runtime $34 + 12n + 23n^2$ can be called $O(n^2)$, since above a large enough value of n, $24n^2$ is always greater
- Informally, we just pick the term which grows fastest relative to n, as this will always become dominant when n gets large enough.

Exercise

Think of an algorithm to check if two strings are anagrams of one another.

Are some methods faster than others?

Use O-notation to try to prove this.

One possible algorithm to check if two strings are anagrams of one another is as follows:

1. Sort the characters in each string.
2. Compare the sorted strings to see if they are equal.

This algorithm has a time complexity of O(n log n), where n is the length of the strings. This is because the time complexity of sorting the strings is O(n log n), and the time complexity of comparing the strings is O(n).

Another possible algorithm is as follows:

1. Create a frequency map for each string, which counts the number of occurrences of each character in the string.
2. Compare the frequency maps to see if they are equal.

This algorithm has a time complexity of O(n), where n is the length of the strings. This is because the time complexity of creating the frequency maps is O(n), and the time complexity of comparing the maps is O(n).

Therefore, the second algorithm is faster than the first algorithm, in terms of time complexity. However, the actual runtime of the algorithms may differ depending on the specific implementation and the input data.

Divide and conquer.

At this point we can begin to establish whether the runtime of an algorithm is $O(n)$ or $O(n^2)$ etc. but we still need to think about cases where it is less obvious how many times a loop will repeat itself.

Divide and conquer algorithms are a design paradigm for solving problems by dividing the problem into smaller subproblems, solving the subproblems recursively, and then combining the solutions to the subproblems to solve the original problem.

Here is a general outline of the divide and conquer approach:

1. Divide: Divide the problem into smaller subproblems.
2. Conquer: Recursively solve the smaller subproblems.
3. Combine: Combine the solutions to the subproblems to solve the original problem.

One of the key advantages of divide and conquer algorithms is that they can often be more efficient than other algorithms, due to their recursive nature. Many problems can be divided into subproblems that are similar to the original problem, and this allows us to reuse solutions and avoid recalculating them.

Examples of divide and conquer algorithms include merge sort, quick sort, and the Karatsuba algorithm for multiplying large integers.

There are some trade-offs to consider when using divide and conquer algorithms. One disadvantage is that they may have a larger constant factor than other algorithms, due to the overhead of dividing and combining the subproblems. Additionally, divide and conquer algorithms are not always the best choice for problems with small input sizes, as the overhead of dividing and combining the subproblems may outweigh the benefits.

Overall, divide and conquer algorithms are a powerful tool for solving problems and can be very efficient for large inputs.

The classic case is the runtime of 'divide and conquer' algorithms. You should have come across the notion of binary search before - but take some time to think about how long it takes to find a name in a telephone directory or a word in a dictionary. You don't have to check every single item in the book. You can focus in on the data you want very quickly. Why is this?

The time it takes to find a name in a telephone directory or a word in a dictionary is much faster than searching through the entire book because both books are organized in a way that allows us to focus in on the data we want very quickly.

Telephone directories and dictionaries are typically organized in alphabetical order, which allows us to use a binary search algorithm to find a specific name or word. Binary search is a divide and conquer algorithm that has a time complexity of O(log n), where n is the number of items in the book. This means that the running time of the algorithm increases very slowly as the size of the input grows, making it much faster than a linear search algorithm, which has a time complexity of O(n).

In a binary search, we start by looking at the middle item in the book. If the item we are looking for comes before the middle item, we can eliminate the second half of the book from our search. If the item we are looking for comes after the middle item, we can eliminate the first half of the book from our search. We then repeat this process on the remaining half of the book until we find the item we are looking for, or until we determine that the item is not in the book.

Because binary search allows us to eliminate half of the items from our search at each step, it can find a specific name or word in a telephone directory or dictionary much faster than a linear search.

Simple binary search

Given a *sorted* collection of items *S* and a value *k* to locate:

- Repeat until *k* is found or no more items exist
    - Check the middle item
    - If it is larger than *k*, remove all items to the right (including the middle item)
    - If it is smaller than *k* remove all items to the left (including the middle item)

How can we express this in O-notation? It's definitely faster than O(*n*) and it isn't a constant O(1).

- How many times can you divide a *n* in half before it becomes less than 1?
- How many times can you multiply 1 by 2 before it becomes greater than *n*?

The algorithm described is a variation of binary search, which is a divide and conquer algorithm that has a time complexity of O(log n), where n is the number of items in the collection S.

In each step of the algorithm, the size of the remaining collection is halved, which means that the running time of the algorithm increases very slowly as the size of the input grows. This is why binary search has a time complexity of O(log n), which is faster than O(n).

To answer your questions:

- You can divide n in half a maximum of log_2(n) times before it becomes less than 1. This is because log_2(n) is the exponent to which we need to raise 2 in order to get n.
- You can multiply 1 by 2 a maximum of log_2(n) times before it becomes greater than n. This is because 2^log_2(n) equals n.

The logarithmic function

The logarithmic function is the inverse of the exponential (power) function. We can think of this in terms of shorthand:

- The addition function is shorthand for the counting function
    - 2 + 4 is the same as starting from 2 and counting up four times
- The multiplication function is shorthand for the addition function
    - 3 * 4 is the same as adding three to itself four times
- The exponential function is shorthand for multiplication
    - 3^4 is three multiplied by itself four times

In fact, these can be generalised into the *Ackermann function*, with shorthand for exponentials, and shorthand for shorthand for exponentials etc. etc.

The logarithmic function is a mathematical function that represents the inverse of the exponential function. It is defined as the exponent to which a fixed base must be raised to produce a given number.

For example, the logarithm of 1000 to base 10 is 3, because 10^3 equals 1000. We can write this as log_10(1000) = 3.

The logarithmic function is commonly used in mathematics and computer science, and it has several important properties:

1. The logarithmic function is an increasing function. This means that if a < b, then log(a) < log(b).
2. The logarithmic function is the inverse of the exponential function. This means that if y = log(x), then x = ey.
3. The logarithmic function satisfies the following property: log(xy) = log(x) + log(y). This means that the logarithm of the product of two numbers is equal to the sum of the logarithms of the numbers.
4. The logarithmic function has the following property: log(x/y) = log(x) - log(y). This means that the logarithm of the ratio of two numbers is equal to the difference of the logarithms of the numbers.

In computer science, the logarithmic function is often used to measure the time complexity of algorithms, because it grows very slowly as the input size increases. For example, an algorithm with a time complexity of O(log n) will generally be much faster than an algorithm with a time complexity of O(n), for large input sizes.

The Ackermann function is a mathematical function that is defined recursively as follows:

Ackermann(m,n) = n+1 if m = 0 Ackermann(m,n) = Ackermann(m-1,1) if m > 0 and n = 0
Ackermann(m,n) = Ackermann(m-1,Ackermann(m,n-1)) if m > 0 and n > 0

The Ackermann function was originally introduced as an example of a well-defined function that is not primitive recursive, meaning that it cannot be computed by a set of basic arithmetic operations and the use of a fixed number of nested loops.

The Ackermann function grows very quickly, even for small input values. For example, the value of Ackermann(4,1) is over 13,000. As a result, the Ackermann function is not used in practice, but it is still studied in theoretical computer science as an example of a function with unusual computational complexity.

One interesting property of the Ackermann function is that it is not total, meaning that there are some input values for which it does not return a result. For example, the function does not return a result for the input values Ackermann(4,2) and Ackermann(5,0). This is because the function is defined recursively, and the recursion goes on indefinitely for some input values, leading to an infinite loop.

Each of the shorthands on the previous sides has an inverse

- Subtraction is the inverse of addition
    - If a + b = c, then c - a = b
- Division is the inverse of multiplication
    - If a * b = c then c / a = b
- The logarithmic function is the inverse of the exponential function
    - If $a^b = c$ then $\log_a c = b$

Logarithmic time

So earlier, when discussing the number of times a binary search repeats itself, we asked the question $2^? = n$.

The answer would be $\log_2 n$ - the base 2 logarithm of n. This function grows very slowly with respect to *n* and so logarithmic time algorithms are extremely desirable.

We can therefore classify binary search (any many other divide and conquer algorithms) as O(log *n*)

In computer science, logarithmic time is a measure of the efficiency of an algorithm, where the running time of the algorithm increases logarithmically with the size of the input.

The logarithmic function is a mathematical function that grows very slowly as the input size increases. For example, the logarithm of a million (1,000,000) to base 2 is just 20, because 2^20 equals 1,000,000. This means that the running time of an algorithm with logarithmic time complexity will increase very slowly as the size of the input grows.

An algorithm with logarithmic time complexity is generally considered to be very efficient, as it can handle large inputs without a significant increase in running time.

An example of an algorithm with logarithmic time complexity is binary search, which is a divide and conquer algorithm used to search for a specific element in a sorted list. Binary search has a time complexity of O(log n), where n is the number of elements in the list. This means that the running time of the algorithm increases very slowly as the size of the input grows.

In general, logarithmic time algorithms are very efficient and are well-suited for problems where the input size may be large. However, they may not be the most efficient choice for problems with small input sizes, due to the overhead of the logarithmic function.

Exercises

Try to implement some of the algorithms discussed in this class, using a programming language of your choice:

- The three problems at the beginning
- An anagram algorithm (preferably fast)
- A divide and conquer algorithm - perhaps to find out the lowest value of *n* for which $a + bn + cn^2$ is less than $(c+1)n^2$
    - Is there a faster way of doing this?

Solutions

1. Given three integers a, b, and c, check if a + b = c:

```
# Initialize variables a, b, and c
a = 2
b = 3
c = 5

# Check if a + b equals c
if a + b == c:
   # If a + b equals c, print "True"
   print("True")
else:
```

```
    # If a + b does not equal c, print "False"
    print("False")
```
This solution has a time complexity of O(1), as it only requires a single comparison operation.

Given a set of integers S = {a, b, c, ...} and an integer k, find out if all elements of S add up to k:

```
# Initialize list S and integer k
S = [1, 2, 3, 4]
k = 10

# Initialize variable sum to 0
sum = 0

# Iterate through elements in S
for i in S:
    # Add element to sum
    sum += i

# Check if sum equals k
if sum == k:
    # If sum equals k, print "True"
    print("True")
else:
    # If sum does not equal k, print "False"
    print("False")
```
This solution has a time complexity of O(n), where n is the number of elements in the set S. It requires one pass through the list S to add up the elements.

2. Given a set of integers S = {a, b, c, ...} and an integer k, find out if any subset of S adds up to k:

```
# Initialize list S and integer k
S = [1, 2, 3, 4]
k = 10
 # Initialize variable found to False
found = False

# Iterate through elements in S
for i in S:
    # Check if element equals k
    if i == k:
        # If element equals k, set found to True and break loop
        found = True
        break
    # If element does not equal k, check if element plus k is in S
    elif (k - i) in S:
        # If element plus k is in S, set found to True and break loop
        found = True
        break

# Print found
print(found)
```
This solution has a time complexity of O(n), where n is the number of elements in the set S. It requires one pass through the list S

Looking at the third problem:

- Given a set of integers *S* and an integer *k*, find out if any subset of S adds up to *k*

Consider a related problem:

- Given two sets of integers, *S* and *W,* and an integer *k*, find out if *W* is a subset of S which adds up to *k*

Is one problem easier to solve than the other? Can you produce an algorithm? What is it's time complexity?

The second problem is generally easier to solve than the first problem, because it only requires us to check if a single subset of S adds up to k, rather than any subset of S.

Here is an algorithm to solve the second problem:

1. Sort the set W in ascending order.
2. Iterate through elements in W.
3. At each element, check if it equals k. If it does, return True.
4. If the element does not equal k, check if k - element is in S. If it is, return True.
5. If the element is not in S and does not equal k, continue to the next element in W.
6. If the end of W is reached, return False.

This algorithm has a time complexity of O(n log n), where n is the number of elements in the set W. This is because the algorithm requires one pass through the sorted list W, and the list W is sorted using a sorting algorithm with time complexity O(n log n).

Here is an example implementation of the algorithm in Python:

```
# Initialize sets S and W and integer k
S = [1, 2, 3, 4]
W = [2, 3]
k = 5

# Sort the set W in ascending order
W.sort()

# Iterate through elements in W
for i in W:
    # Check if element equals k
    if i == k:
        # If element equals k, print "True" and exit loop
        print("True")
        break
    # If element does not equal k, check if k - element is in S
    elif (k - i) in S:
        # If k - element is in S, print "True" and exit loop
        print("True")
        break

# If the end of W is reached, print "False"
print("False")
here is the example implementation again starting from the point where the set W is sorted:
# Sort the set W in ascending order
```

```
W.sort()

# Iterate through elements in W
for i in W:
    # Check if element equals k
    if i == k:
        # If element equals k, print "True" and exit loop
        print("True")
        break
    # If element does not equal k, check if k - element is in S
    elif (k - i) in S:
        # If k - element is in S, print "True" and exit loop
        print("True")
        break

# If the end of W is reached, print "False"
print("False")
```

This code will iterate through the elements in the sorted set W. At each element, it will check if the element equals k or if k - element is in S. If either condition is met, the code will print "True" and exit the loop. If the end of the loop is reached, the code will print "False".

The first problem could be solved by trying every subset of S individually and comparing the sum of each subset to *k*. This would require the evaluation of 2^n sets of integers. This is known as an exponential time algorithm (as the runtime grows exponentially with respect to the input size). Its runtime can be denoted by $O(2^n)$.

The consequences of this are huge - it means that the runtime doubles every time we add an extra element to the set. Build the most powerful computer in the world and all we need to do is find the biggest subset sum problem it can solve using this method, add a handful of extra integers to it and it will grind to a halt again!

We normally regard exponential time algorithms as a bad idea for this reason.

As a consequence, it is generally accepted that not all algorithms are worth using - even algorithms that are totally correct can have such poor time complexities as to be useless in practice.

So, in order to be usable, an algorithm needs not only to get the correct solution but also to get there before it's user dies of old age (and his children and his children's children...). In fact, as we will learn later, we're more likely to use an algorithm which runs quickly with an approximate answer than an exponential time algorithm.

For this reason, we need some notion of an *efficient* solution to a problem, as opposed to a merely accurate one.

Polynomial Time

The working definition which computer scientists use is that an *efficient algorithm* must terminate within *polynomial time*. There are exceptions to this rule, which we will discuss later in the course, but they are quite rare.

In mathematics, a polynomial function is one of the form:

$a + bn + cn^2 + dn^3 + \ldots$ (where n is a variable and $a, b, c, d \ldots$ are all constants)

That is, the addition of a number of terms involving some power of n multiplied by a constant. More intuitively, we can call a polynomial function $O(n^k)$, where k is a constant.

In computer science, polynomial time is a measure of the efficiency of an algorithm, where the running time of the algorithm is a polynomial function of the size of the input.

A polynomial function is a mathematical function in which the highest exponent of the variables is a positive integer. For example, the polynomial function x^2 + 2x + 1 has a degree of 2, because the highest exponent of x is 2.

An algorithm with polynomial time complexity is generally considered to be efficient, as it can handle large inputs without a significant increase in running time. However, algorithms with higher degree polynomial time complexity will generally be slower than algorithms with lower degree polynomial time complexity for large input sizes.

An example of an algorithm with polynomial time complexity is the brute force algorithm for the traveling salesman problem, which has a time complexity of O(n^2 2^n), where n is the number of cities. This means that the running time of the algorithm is a polynomial function of the number of cities, with a degree of 2.

In general, polynomial time algorithms are well-suited for problems where the input size may be large, as they can handle large inputs without a significant increase in running time. However, they may not be the most efficient choice for problems with small input sizes, due to the overhead of the polynomial function.

So algorithms with time complexities such as $O(n)$, $O(n^2)$, $O(\log n)$, $O(n \log n)$ $O(n^3)$ etc are all considered efficient.

In fact, this is a very generous definition, since this would mean that $O(n^{10023})$ is considered theoretically efficient but since solutions like this tend not to appear in practice we don't worry about that too much.

Examples of non-polynomial upper bounds are $O(2^n)$, $O(n!)$, $O(n^n)$ and strictly speaking, $O(n^{\log n})$ since the power to which n is raised may grow with the input size.

Algorithms with runtimes having no polynomial upper bound are to be treated with caution.

The first problem (checking if a + b = c) had an efficient solution, with a time complexity of O(1). This means that the running time of the algorithm does not depend on the size of the input and will always be a constant time.

The second problem (checking if all elements of a set S add up to k) had an efficient solution, with a time complexity of O(n), where n is the number of elements in the set S. This means that the running time of the algorithm increases linearly with the size of the input.

The third problem (checking if any subset of a set S adds up to k) did not have an efficient solution, as it requires checking all possible subsets of S, which has a time complexity of O(2^n), where n is

the number of elements in the set S. This means that the running time of the algorithm increases exponentially with the size of the input, making it impractical for large inputs.

Decision Problems

Both of the problems at the start were *decision problems*. Each instance gives an answer that is either true or false (yes or no). An instance which evaluates to *yes* is a *yes instance* and an instance which evaluates to *no* is a *no instance*.

Consider the following, simple examples. Are they yes or no instances of the subset sum problem?

- The instance {2,3,6,8}, 8

The instance {1,3,4,7}, 13

The first instance {2,3,6,8}, 8 is a "yes" instance of the subset sum problem, because the subset {2,6} adds up to 8.

The second instance {1,3,4,7}, 13 is a "no" instance of the subset sum problem, because there is no subset of the set that adds up to 13.

Question

Is it easier to evaluate a yes instance of subset sum than it is to evaluate a no instance. Think about *both* best and worst case scenarios.

In general, it is easier to evaluate a "yes" instance of the subset sum problem than a "no" instance. This is because a "yes" instance can be determined by finding a single subset that adds up to the target sum, while a "no" instance requires checking all possible subsets and ensuring that none of them add up to the target sum.

In the best case scenario, a "yes" instance can be determined in constant time by finding the subset that adds up to the target sum on the first pass through the input. A "no" instance can also be determined in constant time if the target sum is larger than the sum of all elements in the set.

In the worst case scenario, a "yes" instance can be determined in O(2^n) time, where n is the number of elements in the set, by checking all possible subsets. A "no" instance can also be determined in O(2^n) time by checking all possible subsets and ensuring that none of them add up to the target sum.

Overall, it is generally easier to evaluate a "yes" instance of the subset sum problem than a "no" instance, due to the need to check all possible subsets for a "no" instance.

Other examples: The Travelling Salesperson Problem

Given a set of locations V and a matrix of distances d_{ij} between each pair of locations i and j in V, is it possible to visit all locations exactly once and then return to your starting point while covering a total distance less than k.

You may assume, without loss of generality, that you can start from any one of the locations (it actually makes no difference).

- Think about how to solve a yes instance
- Think about how to solve a no instance

To solve a "yes" instance of the problem, where it is possible to visit all locations exactly once and then return to the starting point while covering a total distance less than k, we can use the following algorithm:

1. Initialize a list of visited locations and a variable total_distance to 0.
2. Iterate through the locations in V.
3. At each location, mark it as visited and add the distance from the previous location to the total distance.
4. If the total distance is greater than or equal to k, return False.
5. If all locations have been visited, return True.

This algorithm has a time complexity of O(n), where n is the number of locations in V, because it requires one pass through the list of locations.

To solve a "no" instance of the problem, where it is not possible to visit all locations exactly once and then return to the starting point while covering a total distance less than k, we can use the following algorithm:

1. Initialize a list of visited locations and a variable total_distance to 0.
2. Iterate through the locations in V.
3. At each location, mark it as visited and add the distance from the previous location to the total distance.
4. If the total distance is greater than or equal to k, return False.
5. If all locations have been visited and the total distance is less than k, return True.

This algorithm has a time complexity of O(n), where n is the number of locations in V, because it requires one pass through the list of locations.

The Python code for the "yes" instance algorithm with detailed line by line comments:

```
# Initialize list of visited locations and variable total_distance to 0
visited = []
total_distance = 0

# Iterate through locations in V
for i in V:
    # Mark location as visited
    visited.append(i)
    # Add distance from previous location to total distance
    total_distance += dij[i][visited[-2]]
    # If total distance is greater than or equal to k, return False
    if total_distance >= k:
        return False

# If all locations have been visited, return True
return True
```

the Python code for the "no" instance algorithm:

```
# Initialize list of visited locations and variable total_distance to 0
visited = []
total_distance = 0

# Iterate through locations in V
for i in V:
    # Mark location as visited
    visited.append(i)
    # Add distance from previous location to total distance
    total_distance += dij[i][visited[-2]]
    # If total distance is greater than or equal to k, return False
    if total_distance >= k:
        return False

# If all locations have been visited and total distance is less than k, return True
if len(visited) == len(V) and total_distance < k:
    return True
else:
    return False
```

Another example

Consider this problem:

Find out if a Boolean expression in *disjunctive normal form* (DNF) is a tautology.

- A tautology is an expression which evaluates to true for all combinations of inputs
    - E.g. $(a \vee \neg a) \wedge (b \vee \neg b)$
- Disjunctive normal form is an *or* of *ands*
    - e.g. $(a \wedge \neg b \wedge c) \vee (a \wedge b \wedge \neg c) \vee \ldots$

Think about yes and no instances again - is it the same as the previous examples?

The problem of determining whether a Boolean expression in disjunctive normal form (DNF) is a tautology is similar to the previous examples of evaluating "yes" and "no" instances. A "yes" instance would be a tautology that evaluates to true for all combinations of inputs, while a "no" instance would be a Boolean expression that does not evaluate to true for all combinations of inputs.

To solve a "yes" instance of the problem, we can iterate through all possible combinations of inputs and check if the Boolean expression evaluates to true for each combination. If the expression evaluates to true for all combinations, we can conclude that it is a tautology.

To solve a "no" instance of the problem, we can also iterate through all possible combinations of inputs and check if the Boolean expression evaluates to true for each combination. If the expression does not evaluate to true for at least one combination, we can conclude that it is not a tautology.

Both solutions have a time complexity of O(2^n), where n is the number of variables in the Boolean expression, because they require checking all possible combinations of inputs.

Witnesses and verification, P, NP and Co-NP

In computer science, the concept of witnesses and verification is often used to determine the complexity of a problem. A witness for a "yes" instance of a problem is a proof or solution that can be used to verify that the instance is a "yes" instance. A witness for a "no" instance of a problem is a proof or counterexample that can be used to verify that the instance is a "no" instance.

The class P (polynomial time) consists of problems for which a "yes" instance can be verified in polynomial time. This means that the time complexity of verifying a solution is a polynomial function of the size of the input. Examples of problems in P include sorting, searching, and matrix multiplication.

The class NP (nondeterministic polynomial time) consists of problems for which a "yes" instance can be verified in polynomial time by using a witness, but it is not known whether a solution can be found in polynomial time. Examples of problems in NP include the traveling salesman problem and the knapsack problem.

The class co-NP consists of problems for which a "no" instance can be verified in polynomial time by using a witness. It is not known whether every "yes" instance of a problem in co-NP has a polynomial time solution.

It is not known whether P = NP, which means it is not known whether every problem in NP can be solved in polynomial time. This is a major open question in computer science and has significant implications for the complexity of certain problems.

- Witnesses and verification: In order to verify a solution to a problem, we need a witness that can be used to prove that the solution is correct. For a "yes" instance of a problem, the witness is a proof or solution that can be used to verify that the instance is a "yes" instance. For a "no" instance of a problem, the witness is a proof or counterexample that can be used to verify that the instance is a "no" instance. Verifying a solution typically requires less time and computational resources than finding a solution.
- P (polynomial time): The class P consists of problems for which a "yes" instance can be verified in polynomial time. This means that the time complexity of verifying a solution is a polynomial function of the size of the input. Examples of problems in P include sorting, searching, and matrix multiplication. Algorithms with polynomial time complexity are generally considered to be efficient, as they can handle large inputs without a significant increase in running time.
- NP (nondeterministic polynomial time): The class NP consists of problems for which a "yes" instance can be verified in polynomial time by using a witness, but it is not known whether a solution can be found in polynomial time. Examples of problems in NP include the traveling salesman problem and the knapsack problem. These problems are considered to be difficult to solve, as finding a solution may require an exponential amount of time and computational resources.

co-NP: The class co-NP consists of problems for which a "no" instance can be verified in polynomial time by using a witness. It is not known whether every "yes" instance of a problem in co-NP has a polynomial time solution. co-NP is the class of problems that are complementary to NP, meaning

that a problem is in co-NP if and only if its complement is in NP. For example, the problem of determining whether a Boolean expression in conjunctive normal form (CNF) is a contradiction is in co-NP, because its complement (determining whether a CNF expression is satisfiable) is in NP.

It is not known whether P = NP, which means it is not known whether every problem in NP can be solved in polynomial time. This is a major open question in computer science and has significant implications for the complexity of certain problems. If P = NP, it would mean that many problems that are currently considered difficult to solve could be solved efficiently, which could have significant impacts on fields such as cryptography and optimization. However, if P ≠ NP, it would mean that there are problems for which it is not known whether efficient solutions exist, which could have implications for the limits of what can be computed in a reasonable amount of time.

Re-cap

Recall the earlier problems:

- Subset sum
- Travelling Salesperson
- Tautology

What did they involve? Can you explain precisely what they are without looking at the slides? Are they potentially easier to resolve in one direction (yes/no) than the other?

Here is a summary of the problems that were mentioned earlier:

- Subset sum: Given a set of integers S and an integer k, find out if any subset of S adds up to k. This problem involves finding a subset of a given set that satisfies a specific condition (adding up to a certain value). It is potentially easier to determine that a "no" instance of the subset sum problem is indeed a "no" instance, because this can be done by simply checking all subsets of the set and seeing if any of them add up to the target value. On the other hand, determining that a "yes" instance is indeed a "yes" instance may require finding a subset that adds up to the target value, which could be more computationally expensive.
- Traveling salesman: Given a set of locations V and a matrix of distances dij between each pair of locations i and j in V, find the shortest possible route that visits all locations exactly once and then returns to the starting point. This problem involves finding the shortest possible route that satisfies a specific condition (visiting all locations exactly once and returning to the starting point). It is potentially easier to determine that a "no" instance of the traveling salesman problem is indeed a "no" instance, because this can be done by simply checking all possible routes and seeing if any of them satisfy the conditions. On the other hand, determining that a "yes" instance is indeed a "yes" instance may require finding the shortest possible route that satisfies the conditions, which could be more computationally expensive.
- Tautology: Given a Boolean expression in disjunctive normal form (DNF), determine if it is a tautology (an expression that evaluates to true for all combinations of inputs). This problem involves determining whether a given Boolean expression satisfies a specific condition (evaluating to true for all combinations of inputs). It is potentially easier to determine that a "no" instance of the tautology problem is indeed a "no" instance, because this can be done by simply checking all possible combinations of inputs and seeing if the expression evaluates

to true for all of them. On the other hand, determining that a "yes" instance is indeed a "yes" instance may require proving that the expression is a tautology, which could be more computationally expensive.

Overall, it may be easier to determine that a "no" instance of a problem is indeed a "no" instance, because this typically involves simply checking all possible solutions and seeing if any of them satisfy the conditions. Determining that a "yes" instance is indeed a "yes" instance may require finding a solution that satisfies the conditions, which could be more computationally expensive. The complexity of determining the answer to a "yes" or "no" instance of a problem can vary depending on the specific problem and the algorithms and approaches used to solve it.

Solution and Verification

Consider two related problems:

- Given an instance of the travelling salesperson problem, find a tour with length less than *k*
- Given an actual travelling salesperson, with an actual route that he/she uses regularly, find out if it is a valid tour with length less than *k*

An algorithm for the first version is said to *solve* the TSP

An algorithm for the second is said to *verify a solution* to TSP given a *witness* (in this case, the tour provided by the actual salesperson).

The first problem involves finding a solution to the traveling salesperson problem, given a set of locations and a maximum allowed length for the tour. This problem is known as the traveling salesperson problem (TSP), and an algorithm that can solve this problem is said to be able to find a tour with a length less than k.

The second problem involves verifying a solution to the traveling salesperson problem, given an actual tour provided by a salesperson. In this case, the tour itself serves as a witness that can be used to verify that it is a valid solution to the TSP (i.e., that it visits all locations exactly once and has a length less than k). An algorithm that can verify a solution to the TSP given a witness is said to be able to verify a solution to the TSP.

Solving the TSP involves finding a tour that satisfies the conditions of the problem, while verifying a solution to the TSP involves checking that a given tour satisfies the conditions of the problem. These are two different tasks, and different algorithms may be used to perform each of them. Solving the TSP may be more computationally expensive than verifying a solution to the TSP, because it may require searching for a tour that satisfies the conditions rather than simply checking a given tour to see if it satisfies the conditions.

P, NP and Co-NP

We already know about the class P. P is the class of problems which can be solved in polynomial time.

The class P consists of problems that can be solved in polynomial time, which means that the time complexity of the algorithm for solving the problem is a polynomial function of the size of the input.

This means that the running time of the algorithm increases at most polynomially with the size of the input, which makes these algorithms efficient and well-suited for handling large inputs. Examples of problems in P include sorting, searching, and matrix multiplication.

NP is the class of problems whose *yes instances* can be verified in polynomial time, given a suitable witness. The N in NP stands for *nondeterministic* and we shall cover its meaning later.

The class NP consists of problems whose "yes" instances can be verified in polynomial time, given a suitable witness. This means that, for a "yes" instance of an NP problem, there exists a proof or solution that can be used to verify that the instance is indeed a "yes" instance, and this verification can be done in polynomial time. However, it is not known whether a solution to an NP problem can be found in polynomial time, so these problems are considered to be difficult to solve. Examples of problems in NP include the traveling salesman problem and the knapsack problem.

Co-NP is the class of problems whose *no instances* can be verified in polynomial time, given a suitable witness. Co just means the complement (or opposite, if you prefer).

The class co-NP consists of problems whose "no" instances can be verified in polynomial time, given a suitable witness. This means that, for a "no" instance of a co-NP problem, there exists a proof or counterexample that can be used to verify that the instance is indeed a "no" instance, and this verification can be done in polynomial time. It is not known whether every "yes" instance of a problem in co-NP has a polynomial time solution. co-NP is the class of problems that are complementary to NP, meaning that a problem is in co-NP if and only if its complement is in NP. For example, the problem of determining whether a Boolean expression in conjunctive normal form (CNF) is a contradiction is in co-NP, because its complement (determining whether a CNF expression is satisfiable) is in NP.

Case Study: Shortest Paths

The shortest path problem is a classic problem in computer science. It asks, given a graph $G = (V, E)$ and a set of distances d_{ij} for each edge (i, j) in E, what is the shortest path between two given nodes in V. These two nodes are often referred to as s (for source) and d (for destination).

The shortest path problem is a classic problem in computer science that asks for the shortest path between two given nodes in a graph, given a set of distances for the edges in the graph. This problem is not a decision problem, as it does not involve determining whether a given instance has a certain property or not. Instead, it involves finding a solution (the shortest path between the two nodes) that satisfies a specific condition (minimizing the total distance).

However, we can make the shortest path problem into an equivalent decision problem by asking whether there exists a path between the two nodes with a total distance less than a given value k. This decision problem can be phrased as: "Given a graph G = (V, E), a set of distances dij for each edge (i, j) in E, and a value k, is there a path between two given nodes s and d in V with a total distance less than k?" This decision problem is a yes/no question, and it can be used to determine whether a given instance of the shortest path problem has a solution that satisfies the condition of having a total distance less than k.

- Is this a decision problem, or something else?

The decision version of the shortest path problem is in NP, because a "yes" instance of the problem can be verified in polynomial time by using a witness (the path itself). However, it is not known whether the problem can be solved in polynomial time, so it is not known whether the shortest path problem is in P. The decision version of the shortest path problem is not in co-NP, because it does not involve verifying "no" instances of the problem.

- If not, can we make it into an equivalent decision problem?
- Is the decision problem in P?
- Is it in NP?
- Is it in Co-NP?

To summarize, the shortest path problem is a problem that involves finding the shortest path between two given nodes in a graph, given a set of distances for the edges in the graph. The problem is not a decision problem, but we can make it into an equivalent decision problem by asking whether there exists a path between the two nodes with a total distance less than a given value k. The decision version of the shortest path problem is in NP, because a "yes" instance of the problem can be verified in polynomial time using a witness (the path itself), but it is not known whether the problem can be solved in polynomial time. The decision version of the shortest path problem is not in co-NP, because it does not involve verifying "no" instances of the problem.

Shortest path decision

The original shortest path problem doesn't have a true or false answer. It requires the output of an actual path between two vertices of a graph.

- In its current form, shortest path can be called an *optimisation problem* or more generally, a *function problem*.
- We can modify it so that it become a decision problem, a bit like last week's version of TSP
    - Instead of asking for the shortest, we can ask if there is a path with total distance less than *k* for some value of *k*. This question has a *yes/no* answer.

We often do this with optimisation problems because it makes them much easier to classify

Optimization problems, like the shortest path problem, do not have a true or false answer and require the output of an actual solution that satisfies a specific condition (in this case, minimizing the total distance). By modifying the problem to ask whether there exists a path with a total distance less than a given value k, we can turn it into a decision problem, which has a yes/no answer. This can be useful because decision problems are easier to classify and can be more efficiently solved using algorithms and data structures designed for decision problems.

It is often the case that optimization problems can be transformed into decision problems by asking whether a solution exists that satisfies a certain condition. For example, in the case of the traveling salesman problem, we can ask whether there exists a tour with a total distance less than a given

value k, rather than asking for the shortest possible tour. This transformation can make it easier to classify the problem and find efficient algorithms for solving it.

Shortest path problem classification

Is shortest path in P?

YES: run Dijkstra's algorithm and if the shortest path is less than *k*, return true. Otherwise return false.

Is it in NP?

YES: given a path from *s* to *d* as a witness, we can easily check it's a valid path and add up the edge distances to see if the total is less than *k* (alternatively we can just ignore the witness and use Dijkstra's algorithm)

Is it in Co-NP? YES: simply ignore the witness and run Dijkstra's algorithm

It is not known whether the shortest path problem is in P, which is the class of problems that can be solved in polynomial time. The shortest path problem involves finding the shortest path between two given nodes in a graph, given a set of distances for the edges in the graph. While the decision version of the shortest path problem, which asks whether there exists a path with a total distance less than a given value k, is in NP (nondeterministic polynomial time), it is not known whether the problem can be solved in polynomial time.

NP is the class of problems whose yes instances can be verified in polynomial time, given a suitable witness. In the case of the shortest path problem, a witness would be the path itself. However, finding a solution to the shortest path problem may require more time and computational resources than simply verifying a solution, so it is not known whether the problem is in P. It is possible that the shortest path problem is NP-hard, which means that it is at least as difficult to solve as the hardest problems in NP.

Dijkstra's algorithm is a graph search algorithm that solves the single-source shortest path problem for a graph with nonnegative edge weights, producing a shortest path tree. This algorithm is often used in routing and as a subroutine in other graph algorithms. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.

Here is an example implementation of Dijkstra's algorithm in Python, with line-by-line comments:

```
import math

# Helper function to find the index of the vertex with the minimum distance
def find_min_dist(distance, visited):
    min_distance = math.inf
    min_index = None
    for i, dist in enumerate(distance):
        if dist < min_distance and i not in visited:
            min_distance = dist
            min_index = i
    return min_index
```

```
# Function to perform Dijkstra's algorithm
def dijkstra(graph, source):
    # Number of vertices in the graph
    n = len(graph)
    # Initialize distances and visited lists
    distance = [math.inf] * n
    visited = []
    # Set the distance of the source vertex to 0
    distance[source] = 0
    # Iterate n - 1 times
    for _ in range(n - 1):
        # Find the vertex with the minimum distance
        min_index = find_min_dist(distance, visited)
        # Mark the vertex as visited
        visited.append(min_index)
        # Update the distances of the neighboring vertices
        for i, dist in enumerate(graph[min_index]):
            if dist != 0 and i not in visited:
                distance[i] = min(distance[i], distance[min_index] + dist)
    # Return the distance list
    return distance

# Test the function with a graph represented as an adjacency matrix
graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0],
         [4, 0, 8, 0, 0, 0, 0, 11, 0],
         [0, 8, 0, 7, 0, 4, 0, 0, 2],
         [0, 0, 7, 0, 9, 14, 0, 0, 0],
         [0, 0, 0, 9, 0, 10, 0, 0, 0],
         [0, 0, 4, 14, 10, 0, 2, 0, 0],
         [0, 0, 0, 0, 0, 2, 0, 1, 6],
         [8, 11, 0, 0, 0, 0, 1, 0, 7],
         [0, 0, 2, 0, 0, 0, 6, 7, 0]]

# The source vertex is vertex 0
source = 0
# Get the distance list for the graph
distances = dijkstra(graph, source)
# Print the distance from the source vertex to each other vertex
print(distances)
```

This code will initialize a distance list with infinities and a visited list with no vertices. It will then set the distance of the source vertex to 0 and iterate n - 1 times, where n is the number of vertices in the graph. At each iteration, it will find the vertex with the minimum distance that has not yet been visited and mark it as visited. It will then update the distances of the neighbouring vertices by taking the minimum of the current distance and the distance through the current vertex. Finally, it will return the distance list.

General result

In fact, we can say with certainty that all problems in P are also in both NP and Co-NP. Since both yes and no instances can be solved (and therefore verified) efficiently.

However, we must ask an important question at this stage:

Are there some problems which are in NP but not in P?

IT'S WORTH A MILLION DOLLARS IF YOU CAN ANSWER IT

Yes, there are problems that are in NP but not known to be in P. These are known as NP-complete problems. NP-complete problems are a class of problems that are at least as hard as any problem in NP, and if any NP-complete problem can be solved in polynomial time, then all problems in NP can be solved in polynomial time. This means that if we can find a polynomial-time algorithm for any NP-complete problem, we can use it to solve all problems in NP in polynomial time as well. Some examples of NP-complete problems include the traveling salesman problem, the knapsack problem, and the Boolean satisfiability problem.

Subset Sum

We can say that subset sum is in NP:

- Given a subset of numbers from *S*, we can check that they add up to *k*

However, there is no *known* algorithm to solve subset sum in polynomial time. We simply do not know if it is in P.

Perhaps more interestingly: the above verification process relies on the existence of a subset which adds up to *k*. This means that it can only work with *yes* instances of the problem. There is no *known* equivalent procedure to check *no* instances of the problem. We simply do not know if subset sum is in Co-NP either!

The subset sum problem is in NP because we can verify a "yes" instance (i.e. a subset that adds up to the target value) in polynomial time by checking the sum of the subset. However, it is not known whether the problem can be solved in polynomial time, so it is not known whether it is in P. Additionally, as you mentioned, there is no known equivalent procedure to verify "no" instances of the problem, so it is not known whether the subset sum problem is in co-NP. This means that the subset sum problem is an example of a problem that is in NP but not known to be in P or co-NP.

Tautology

In logic and mathematics, a tautology is a formula that is always true, regardless of the values of its variables. For example, the formula "A or not A" is a tautology, because it is true no matter what value A takes. In computer science, the problem of determining whether a Boolean expression in disjunctive normal form (DNF) is a tautology is known as the tautology problem.

The tautology problem involves evaluating a given Boolean expression for all possible combinations of inputs and determining whether the expression always evaluates to true. For example, given the Boolean expression $(A \vee \neg A) \wedge (B \vee \neg B)$, we can evaluate the expression for all possible combinations of A and B and check whether it is always true. In this case, the expression is a tautology, because it is true for all combinations of A and B.

The tautology problem is an example of a decision problem, because it involves determining whether a given instance has a certain property (being a tautology) or not. It is potentially easier to determine that a "no" instance of the tautology problem is indeed a "no" instance, because this can be done by simply checking all possible combinations of inputs and seeing if the expression ever evaluates to false. On the other hand, determining that a "yes" instance is indeed a "yes" instance

may require evaluating the expression for all possible combinations of inputs, which could be more computationally expensive.

Similarly, we can say that tautology is in Co-NP:

- Given a suitable truth assignment for the variables in the formula, we can check that it evaluates to false.

However, there is no *known* algorithm to solve tautology in polynomial time. We simply do not know if it is in P.

Again: the above verification process relies on the existence of a truth assignment which evaluates to false. This means that it can only work with *no* instances of the problem. There is no *known* equivalent procedure to check *yes* instances of the problem. We do not know if tautology is in NP!

It is worth noting that, while we do not know if tautology is in NP, we do know that it is in the larger class of problems known as PSPACE (polynomial space). This means that it can be solved using a polynomial amount of memory, even if it may require an exponential amount of time to solve.

Food for thought...

This problem (like many in compsci) is related to Tautology.

Given any Boolean circuit *C*, is there a circuit with equivalent output using *k* gates or less.

Is it in P?

Is it in NP?

Is it in Co-NP?

You probably used to solve (very) small instances of this as an undergraduate, using Karnaugh Maps - but think very carefully about the general case.

This problem is related to tautology in that it involves determining whether a given Boolean circuit has a certain property (using k gates or fewer). However, it is a different problem, and its complexity class is not necessarily the same as tautology.

It is not clear whether this problem is in P, as it is not known whether there exists an algorithm that can solve it in polynomial time. Similarly, it is not known whether this problem is in NP, as it is not known whether there exists a witness that can be used to verify a "yes" instance of the problem in polynomial time. It is also not known whether this problem is in co-NP, as it is not known whether there exists a witness that can be used to verify a "no" instance of the problem in polynomial time.

NP-Completeness and NP-Hardness

NP-Completeness

The next question we need to address is “how can I tell if my problem can be solved efficiently?” We’ve had a go at writing algorithms for simple problems and found some we can’t solve efficiently - but is there a way to tell if there’s no chance of finding an efficient solution?

One way to determine if a problem can be solved efficiently is to try to find an algorithm that solves the problem in polynomial time. If such an algorithm exists, then the problem is considered to be in the class P, which represents problems that can be solved efficiently. If it is not possible to find an algorithm that solves the problem in polynomial time, then the problem might still be solvable, but it may take a longer amount of time and computational resources to find a solution.

Another way to determine the efficiency of a problem is to consider whether the problem is in NP, which represents problems whose solutions can be verified in polynomial time. If a problem is in NP, it means that it is possible to verify a solution to the problem quickly, even if finding a solution may take longer. However, if a problem is not in NP, it means that it is not possible to verify a solution to the problem efficiently, even if a solution can be found quickly.

Finally, some problems may be in co-NP, which represents problems whose non-solutions can be verified in polynomial time. If a problem is in co-NP, it means that it is possible to quickly verify that a given solution is not valid, even if finding a valid solution may take longer.

Overall, it is often difficult to determine the efficiency of a problem, as it may not be clear whether an efficient solution exists or not. It is important to continue researching and testing different approaches to solve a problem, in order to determine the most efficient way to solve it.

The answer is, sort of!

If we can show that a problem is *NP-Complete* (or harder) it means that there is no known efficient solution to it. We can't prove that there never will be but we can say that it is beyond the realms of efficient computation at present.

NP-Complete problems are a special class of NP problems that are at least as hard as any other problem in NP. In other words, if we can find an efficient solution to an NP-Complete problem, then we can also use that solution to solve all other problems in NP efficiently. This makes NP-Complete problems the "hardest" problems in NP.

To show that a problem is NP-Complete, we need to prove that it is both in NP and that it is at least as hard as any other problem in NP. This is usually done by demonstrating a polynomial-time reduction from the problem to an NP-Complete problem.

If a problem is in P, then it can be solved efficiently and we don't need to worry about whether it is NP-Complete or not. If a problem is in NP but not known to be in P, then it might be possible to find an efficient solution, but we don't know for sure. If a problem is NP-Complete, then it is definitely not known to be solvable efficiently, and it is unlikely that an efficient solution will be found in the near future.

If a problem is NP-Complete, it means that finding an efficient solution to that problem is equivalent to finding an efficient solution to all of the problems in NP. Additionally, the problem must be a member of NP itself. The NP-Complete problems are the hardest of all problems in NP. Subset sum and Travelling Salesperson are NP-Complete, amongst many others.

To show that a problem is NP-Complete, we can use a process called "reduction". Reduction is the process of taking an instance of an NP-Complete problem and transforming it into an instance of

another problem, without changing the answer. If we can find a way to transform an instance of an NP-Complete problem into an instance of our problem, and we can transform the solution of our problem back into a solution of the NP-Complete problem, then we can say that our problem is at least as hard as the NP-Complete problem. If we can show that our problem is at least as hard as every problem in NP, then we can say that our problem is NP-Complete.

The first problem to have been proved NP-Complete was the boolean satisfiability (SAT) problem. It was proved NP-Complete by Stephen Cook in 1971, the proof being known as Cook's Theorem or the Cook-Levin Theorem.

The boolean satisfiability problem asks whether it is possible to assign values to a set of variables in a boolean formula such that the formula evaluates to true. For example, given the formula (a ∨ ¬b ∧ c), we might ask whether it is possible to assign values to a, b, and c such that the formula evaluates to true. This problem is important because many problems in computer science can be reduced to the boolean satisfiability problem.

To prove that a problem is NP-Complete, we must first show that it is a member of NP. This means that we need to be able to verify a "yes" instance of the problem in polynomial time. We then need to find a way to reduce an instance of any other problem in NP to an instance of the problem we are trying to prove is NP-Complete. This reduction must be done in polynomial time, and it must preserve the answer to the problem (i.e., if the original problem is a "yes" instance, the reduced problem must also be a "yes" instance, and vice versa). If we can do this, we can say that the problem is NP-Complete.

It involves transforming the description of a nondeterministic Turing machine into an instance of SAT. The proof is not greatly complex but it is long and requires an explanation of nondeterministic Turing machines.

Satisfiability (SAT)

Cook's Theorem demonstrates that *any* NP problem can be transformed into an instance of SAT in polynomial time.

a brief overview of the proof:

First, we need to understand what it means for a problem to be in NP. A problem is in NP if there exists a polynomial-time algorithm for verifying solutions to the problem, given a witness. In other words, if we are given a solution to the problem and a witness, we can check whether the solution is correct in polynomial time.

The SAT problem is defined as follows: given a Boolean formula in conjunctive normal form (CNF), determine whether there exists a truth assignment for the variables in the formula such that the formula evaluates to true. This problem is a decision problem, because it involves determining whether a given instance has a certain property (a truth assignment that makes the formula evaluate to true).

Cook's Theorem states that if any problem in NP can be reduced to the SAT problem in polynomial time, then SAT is NP-Complete. In other words, if we can find a polynomial-time algorithm for

transforming an instance of any problem in NP into an instance of SAT, and if we can solve the resulting SAT instance in polynomial time, then SAT is NP-Complete.

To prove Cook's Theorem, we need to show that the SAT problem is in NP and that any problem in NP can be reduced to the SAT problem in polynomial time. The fact that SAT is in NP follows from the definition of NP, as we can verify solutions to the SAT problem in polynomial time using a witness (the truth assignment). To show that any problem in NP can be reduced to the SAT problem in polynomial time, we need to use the concept of a nondeterministic Turing machine.

A nondeterministic Turing machine is a theoretical model of computation that can "guess" the next step in its computation. This allows it to solve problems in NP, because it can try different guesses and check their validity in polynomial time. Using a nondeterministic Turing machine, we can transform any problem in NP into an instance of SAT in polynomial time, by encoding the computation of the Turing machine as a Boolean formula. This completes the proof of Cook's Theorem.

SAT asks the question, given a boolean formula in *conjunctive normal form* (an *and* of *or*s) does *any* combination of inputs evaluate to *true*. Such a combination is called a *satisfying assignment*.

- e.g. $(a \vee \neg b \vee c) \wedge (a \vee b \vee \neg c) \wedge \ldots$

- For example, consider the formula (a ∨ ¬b) ∧ (¬a ∨ b). This formula has two satisfying assignments: {a=True, b=True} and {a=False, b=False}.
- To prove that SAT is NP-Complete, Cook first showed that it is a member of NP. To do this, he showed that given a formula and a proposed satisfying assignment, it can be verified in polynomial time whether the assignment is indeed a satisfying assignment for the formula.
- Cook then showed that if any problem in NP can be efficiently reduced to SAT, then SAT is NP-Complete. In other words, if there is an efficient way to transform instances of any NP problem into instances of SAT, and if there is an efficient way to transform solutions to those SAT instances back into solutions for the original problem, then SAT is NP-Complete. This is known as the NP-Completeness of SAT.
- Since the proof of Cook's Theorem, many other problems have been shown to be NP-Complete, including subset sum and the traveling salesman problem.

We say that all problems in NP *reduce* to SAT in polynomial time, and this can be donated by the symbol $\geq_P$. As in NP $\geq_P$ SAT

This means that any problem in NP can be transformed into an instance of the SAT problem in polynomial time. Additionally, if there is a polynomial-time solution to the SAT problem, then there is a polynomial-time solution to all problems in NP. This is why the SAT problem is considered to be the "hardest" problem in NP. If we can find an efficient solution to the SAT problem, then we can use it to solve all of the other problems in NP efficiently as well.

Consequences of NP-Completeness

We need to think about this result carefully. If there is a polynomial time process which converts any instance of problem *A* into an equivalent instance of problem *B* then a polynomial time algorithm for *B* would give us a polynomial time solution for problem *A*.

This is known as the "reduction principle," and it is a very important concept in computer science. It allows us to understand the relationship between different problems and how they relate to each other in terms of computational complexity. In the case of NP-Completeness, it means that if we can find an efficient solution to an NP-Complete problem, then we can use that solution to solve all of the other problems in NP efficiently as well. This is a very powerful result, as it tells us that if we can find an efficient solution to any NP-Complete problem, then we can solve all of the other problems in NP efficiently as well.

This means that if we were to find a polynomial time solution to SAT, this would give us a polynomial time solution to every problem in NP.

This is why finding a polynomial time solution to SAT, or any other NP-Complete problem, is considered to be a major breakthrough in computer science. It would mean that all of the problems in NP could be solved efficiently, which would have far-reaching consequences in many fields that rely on computational complexity. However, it is important to note that it is not known whether or not this is actually possible, and it is one of the most fundamental open questions in computer science.

This would include problems we don't want to solve such as integer factorisation (which would enable us to hack the RSA encryption system).

Thus, if we can find an efficient solution to SAT, this would have major consequences for computer science and cryptography. This is why showing that a problem is NP-Complete (like SAT) is considered to be a major result. It implies that the problem is at least as difficult as all other problems in NP, and thus finding an efficient solution to the problem would have far-reaching consequences.

SAT is not the only NP-Complete problem, let's look at some more...

Some other examples of NP-Complete problems include:

- 3-SAT: A variant of SAT where each clause contains at most 3 variables.
- Hamiltonian cycle: Given a graph, find a cycle that visits every vertex exactly once.
- Knapsack problem: Given a set of items with weights and values, find the maximum total value of items that can be included in a knapsack with a given weight capacity.
- Subset sum: Given a set of integers S and an integer k, find out if any subset of S adds up to k.
- Traveling salesman problem: Given a set of locations V and a matrix of distances dij between each pair of locations i and j in V, find the shortest possible route that visits all locations exactly once and then returns to the starting point.

It is important to note that there may be efficient solutions for specific instances of these problems, but there is no known general solution that works for all instances in polynomial time.

Satisfiability $\geq_P$ 3-SAT

Having established that satisfiability is NP-Complete, we can show that 3-SAT is. 3-SAT is the version of SAT where there are only ever three literals per clause.

To show that 3-SAT is NP-Complete, we use the fact that SAT is NP-Complete and show that 3-SAT can be reduced to SAT in polynomial time. This means that if we had a polynomial time solution for 3-SAT, we could use it to solve SAT in polynomial time, and therefore all problems in NP.

To reduce SAT to 3-SAT, we can take any boolean formula in conjunctive normal form and transform it into an equivalent formula in 3-SAT form by adding auxiliary variables and clauses as needed. This process can be done in polynomial time, so we have shown that 3-SAT is NP-Complete.

Other examples of NP-Complete problems include the knapsack problem, the partition problem, and the vertex cover problem. These problems are all known to be at least as hard as any problem in NP, and no efficient solutions are known for them.

So if we take a general SAT formula, each clause has either less than 3, exactly 3, or more than 3 literals in it.

- If it has exactly 3, do nothing
- If it has less than 3, simply repeat one of the literals (it makes absolutely no difference to the truth assignment)
- If it has more than three...
- We can split the clause into smaller clauses, each with exactly three literals. For example, the clause (A ∨ B ∨ C ∨ D ∨ E) can be split into the following three clauses: (A ∨ B ∨ C), (A ∨ B ∨ D), and (A ∨ B ∨ E). This process is known as reduction.
- By showing that any instance of SAT can be reduced to an instance of 3-SAT in polynomial time, we can conclude that 3-SAT is NP-Complete.
- Similarly, we can show that other problems are NP-Complete by reducing them to other NP-Complete problems. This process is known as showing NP-hardness. For example, the subset sum problem is NP-Hard because it can be reduced to 3-SAT in polynomial time.

Satisfiability $\geq_P$ 3-SAT

If a clause has more than three literals:

- e.g. $(a \vee \neg b \vee c \vee \neg d \vee e \vee \neg f)$
- We can turn it into a series of 3-literal clauses with the use of some extra literals ***l***
- $(a \vee \neg b \vee l_1) \wedge (\neg l_1 \vee c \vee l_2) \wedge (\neg l_2 \vee \neg d \vee l_3) \wedge (\neg l_3 \vee e \vee \neg f)$

This process is called reduction, and it allows us to turn any instance of SAT into an equivalent instance of 3-SAT in polynomial time. Therefore, we can say that SAT ≥P 3-SAT, which means that 3-SAT is also NP-Complete.

Now, we can use this process of reduction to show that many other problems are NP-Complete as well. Essentially, if we can show that a problem A reduces to SAT in polynomial time, and we already know that SAT is NP-Complete, then we can conclude that problem A is also NP-Complete. This is a powerful tool for showing that many problems are NP-Complete, and it is the basis for the concept of NP-Completeness.

It should be clear that this increases the total size of the problem by no more than a factor of 3 (so the transformation is polynomial time)

It should also be clear that any satisfying assignment for the original clause will also be a satisfying assignment for the transformed 3-SAT formula, and vice versa. This means that the 3-SAT problem is at least as hard as the SAT problem, which we already know to be NP-Complete. Therefore, 3-SAT is also NP-Complete.

Since it should be plain that 3-SAT is in NP, then 3-SAT is NP-Complete.

We can show that 3-SAT is NP-complete by proving that it is in NP (a "yes" instance can be verified in polynomial time by using a witness, which is a satisfying assignment) and that it is at least as hard as any other problem in NP. We do this by showing that we can transform any other problem in NP into an equivalent instance of 3-SAT in polynomial time. This means that if we can find an efficient solution to 3-SAT, we can use it to solve all other problems in NP efficiently as well.

3-SAT $\geq_P$ Independent Set

The independent set problem asks, given a graph $G = (V, E)$, if there is a subset S of V such that no two nodes of S are adjacent and $|S| > k$.

It is unknown whether the independent set problem is in P, NP, or co-NP. It is not known whether it is possible to find an independent set of a given size in polynomial time, or whether it is possible to verify the existence of an independent set of a given size in polynomial time. The independent set problem is one of many problems that are not known to be in any of the classes P, NP, and co-NP.

For any instance of 3-SAT, we can easily produce an equivalent instance of independent set, whereby the 3-SAT formula is satisfiable *iff* there is an independent set of size n.

To do this, we create a graph where each variable in the 3-SAT formula is represented by two nodes in the graph (one for each possible value). Then, for each clause in the 3-SAT formula, we connect the two nodes representing the negation of each literal in the clause. This means that if a node representing a literal is included in the independent set, then the negation of that literal cannot be included. Therefore, if we can find an independent set of size n (where n is the number of variables in the 3-SAT formula), then we have found a satisfying assignment for the 3-SAT formula.

Since independent set is in NP (a "yes" instance can be verified in polynomial time by using a witness, which is the independent set itself), and it can be reduced to 3-SAT in polynomial time, independent set is also NP-Complete.

First, we arrange each clause into a triangle of nodes and label each of them with their literals, then we connect any two opposites (such as a and $\neg a$ with an edge).

We can then use this graph to check for the existence of an independent set. If there is an independent set of size n (where n is the number of clauses in the 3-SAT formula), this means that there exists a truth assignment that satisfies all of the clauses, because no two nodes in the independent set are connected by an edge. Conversely, if the 3-SAT formula is satisfiable, this means that there exists a truth assignment that satisfies all of the clauses, and this truth assignment can be represented by an independent set of size n in the graph. Therefore, the 3-SAT problem is

polynomial-time reducible to the independent set problem, which means that the independent set problem is NP-hard.

It is not known whether the independent set problem is in NP, so it is not known whether it is NP-Complete. However, it is believed to be NP-hard, which means that it is at least as hard as the hardest problems in NP.

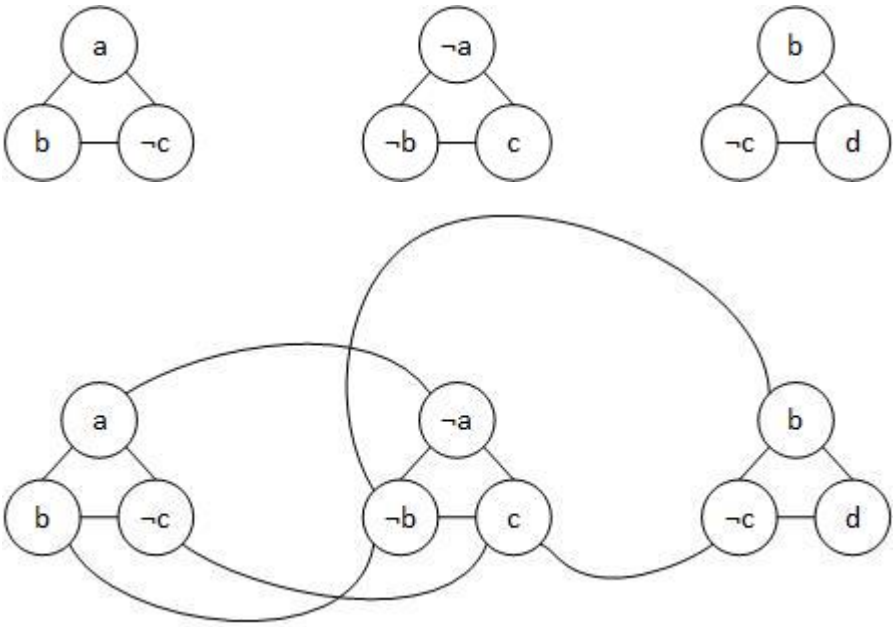
An example follows...

3-SAT $\geq_P$ *Independent Set*

First create triangles:

Then connect opposites:

If there is a satisfying assignment, it will be possible to select one member of *S* from each clause without it being connected to another member of *S*.



3-SAT $\geq_P$ Vertex Cover

The vertex cover problem (in decision form) asks, given a graph $G = (V, E)$ is there a subset S of V such that every edge in E is incident on a vertex of S and $|S| < k$.

To show that the independent set problem is NP-Complete, we can reduce it to the vertex cover problem. The vertex cover problem is defined as follows: given a graph G = (V, E), find a subset S of V such that for every edge (u, v) in E, at least one of u or v is in S, and |S| is minimized.

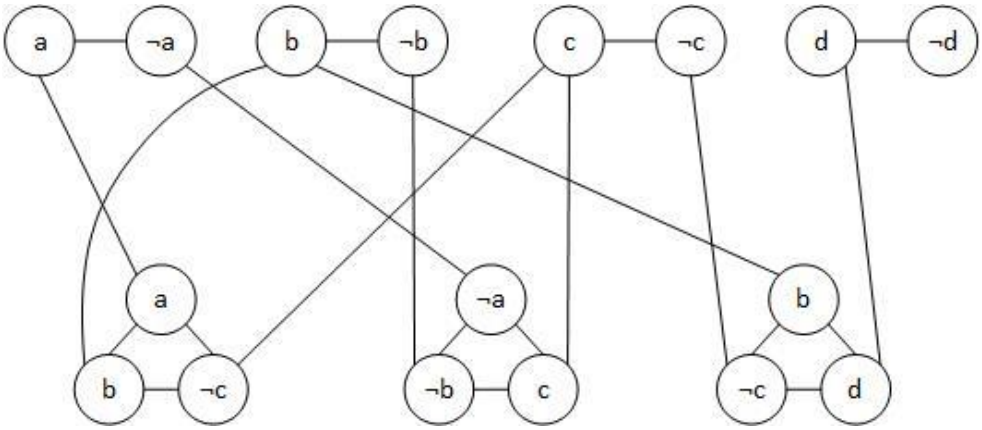
To reduce the independent set problem to the vertex cover problem, we can consider the complement of the given graph G, which is a graph G' = (V', E') where V' = V and E' = {(u, v) | (u, v) is not in E}. Then, we can find a vertex cover in G' by finding an independent set in G, and the size of the independent set in G will be equal to the size of the vertex cover in G'.

Since the vertex cover problem is known to be in NP, this reduction shows that the independent set problem is also in NP. Thus, the independent set problem is NP-Complete.

For each variable *v* in our 3-SAT formula, we create two nodes and label them *v* and ¬*v* and join them with an edge.

(*a* ∨ *b* ∨ ¬*c*) ∧ (¬*a* ∨ ¬*b* ∨ *c*) ∧ (*b* ∨ ¬*c* ∨ *d*)

Then for each clause, we produce a node for each literal and connect them in a triangle. Finally connect nodes with the same label.



We know that at least one of the node pairs along the top must be in the vertex cover (or else the edges in between won't be covered). Also, at least 2 out of 3 of the nodes in the triangles at the bottom must be in the cover.

So, if we can find a vertex cover of size k, we can use this to satisfy the 3-SAT formula by setting the variables in the covered nodes to true and the ones in the uncovered nodes to false. This means that the independent set problem is NP-Complete.

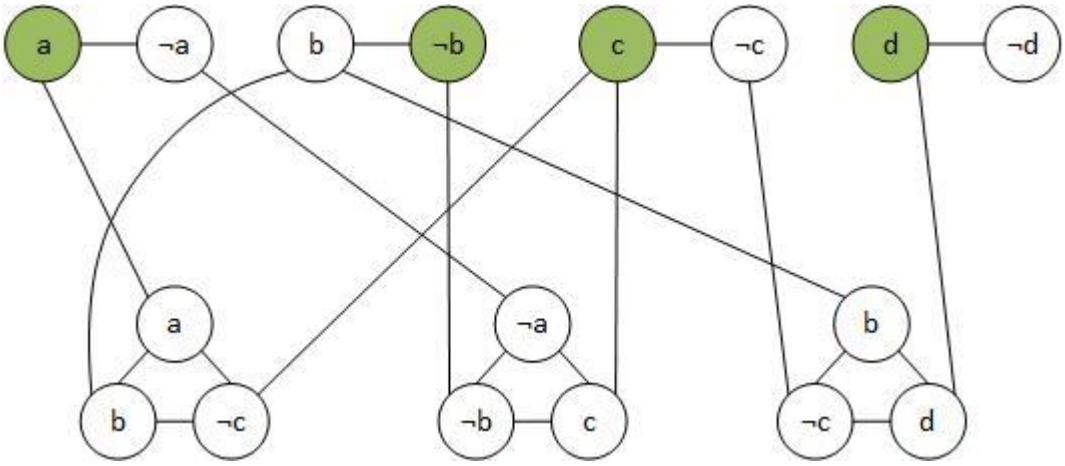
The vertex cover problem is the opposite of the independent set problem, it asks whether there is a subset of nodes in a graph such that all edges are incident to at least one node in the subset. It is easy to see that the vertex cover problem is in NP, because a "yes" instance (a vertex cover) can be verified in polynomial time by simply checking that all of the edges are incident to at least one node in the subset. Since the independent set problem reduces to the vertex cover problem, it follows that the vertex cover problem is also NP-Complete.

It turns out that if the formula is not satisfiable, more vertices need to be added to the cover - and if it isn't they don't.

This means that the size of the vertex cover is equal to the number of variables in the 3-SAT formula plus a constant if the formula is not satisfiable, and equal to the number of variables plus another constant if the formula is satisfiable. This means that the independent set problem is in NP, and therefore it is NP-Complete.

So the formula is satisfiable *iff* there is a vertex cover with size $n+2c$ or less, where n is the number of variables and c the number of clauses.

This means that the vertex cover problem is NP-Complete, because it is in NP (a "yes" instance of the problem can be verified in polynomial time by using a witness, which is the vertex cover itself) and it is NP-hard (every problem in NP can be reduced to it in polynomial time).



NP-Complete Problems

So we have demonstrated that Satisfiability, 3-SAT, Independent Set and Vertex Cover are all NP-Complete. If we can find an efficient solution to any one of them, we can efficiently solve every problem in NP.

This means that if a problem is NP-Complete, it is considered to be a hard problem, because any algorithm for solving it must be able to solve all problems in NP in polynomial time. NP-Complete problems are considered to be the hardest problems in NP, and finding efficient solutions to them is a major open problem in computer science.

In fact there is an enormous, growing number of known NP-Complete problems covering a range of applications from computer science, operations research, mathematics, biology, chemistry and many more subject areas. Even puzzles like tetris, minesweeper and candy crush have been proven NP-Complete.

It's important to note that the fact that a problem is NP-Complete does not mean that it is necessarily difficult to solve in practice. Many NP-Complete problems can be solved relatively efficiently for small input sizes, and there may be efficient heuristics or approximate algorithms that work well for larger input sizes. However, there is no known way to solve NP-Complete problems in polynomial time, so they become increasingly difficult to solve as the input size grows.

We haven’t found a way to solve any of them efficiently - and we’ve had over 40 years to work on it. Neither can we prove that no efficient solution exists!

However, it is important to note that NP-Completeness is not a guarantee that a problem cannot be solved efficiently. It is simply a theoretical result that tells us that if an efficient solution to an NP-Complete problem were to be found, then this would also give us an efficient solution to all other problems in NP. In practice, it is possible that some NP-Complete problems may turn out to have efficient solutions, but this has not yet been demonstrated for any such problem.

Proof techniques

- Restriction
    - This is the easiest and most common: show that a special case of your problem is NP-complete.
    - A good example is reducing vertex cover to weighted vertex cover.
- Local Replacement
    - Make a change to each component of your problem so that it becomes an equivalent component of the new problem
    - Example SAT to 3-SAT
- Component Design
    - The hard way - manufacture components which enforce equivalence between instances
    - Example 3-SAT to Vertex cover

Transformation ○ Convert a known NP-complete problem into your problem. ○ A good example is reducing 3-SAT to independent set. ● Completeness ○ Show that every problem in NP reduces to your problem in polynomial time. ○ This is very rare and is only known for a handful of problems.

To prove that a problem is NP-complete, it is sufficient to show that it is in NP and that it can be reduced to a known NP-complete problem in polynomial time. This is often done by using one of the proof techniques mentioned above. It is also possible to prove that a problem is NP-hard, which means that it is at least as hard as the hardest problems in NP, even if it is not necessarily in NP itself.

Practice

Given that the Hamiltonian cycle problem is NP-Complete

- Given a graph $G = (V,E)$, is there a cycle which visits each node exactly once (a cycle must return to its starting point)

The Hamiltonian cycle problem is a decision problem that involves determining whether a given graph has a cycle that visits each node exactly once. It is known to be NP-Complete, which means that it is not known whether there is an efficient algorithm for solving the problem. This means that finding a solution to the Hamiltonian cycle problem is equivalent to finding an efficient solution to all of the problems in NP. The Hamiltonian cycle problem is in NP because a "yes" instance of the problem (i.e., a graph with a Hamiltonian cycle) can be verified in polynomial time by using a witness (the cycle itself). However, it is not known whether the problem can be solved in polynomial time, so it is not known whether the Hamiltonian cycle problem is in P. The Hamiltonian cycle problem is not in co-NP, because it does not involve verifying "no" instances of the problem.

Prove the following are also NP-Complete

- The longest simple cycle problem
    - Given a graph $G = (V,E)$, is there a cycle which visits k nodes or more?
- The travelling salesman problem
    - You know this one...

To prove that the longest simple cycle problem is NP-Complete, we can use a reduction from the Hamiltonian cycle problem. Given a graph G and an integer k, we can create a new graph G' by adding k-1 new nodes to G and connecting them to all of the nodes in G with edges of length 1. If G has a Hamiltonian cycle, then this cycle must pass through all of the nodes in G, and it will also pass through the k-1 new nodes because they are connected to all of the nodes in G. Thus, G' will have a simple cycle of length k or more. On the other hand, if G' has a simple cycle of length k or more, then this cycle must pass through all of the nodes in G, so G must have a Hamiltonian cycle. This reduction can be done in polynomial time, so the longest simple cycle problem is NP-Complete.

To prove that the traveling salesman problem is NP-Complete, we can use a reduction from the Hamiltonian cycle problem. Given a graph G, we can create a new graph G' by adding a new node s and connecting it to all of the nodes in G with edges of length 1. If G has a Hamiltonian cycle, then we can use this cycle as a solution to the traveling salesman problem in G' by visiting all of the nodes in G and then returning to s. On the other hand, if we have a solution to the traveling salesman problem in G', then this solution must visit all of the nodes in G and return to s, which means that G must have a Hamiltonian cycle. This reduction can also be done in polynomial time, so the traveling salesman problem is NP-Complete.

Moving forward - Optimisation Problems

Optimisation problems involve finding the minimum or maximum value of a function. Consequently they are not in either P or NP as they are not decision problems so they cannot be called NP-Complete.

However, if we transform the optimisation problem into an equivalent decision problem by asking whether or not the function has a value greater/less than some constant *k*, we can prove that this problem is NP-Complete.

However, we can still try to find an efficient solution for them, or we can try to transform them into an equivalent decision problem so that we can use NP-Completeness results to classify them. For example, the traveling salesman problem is an optimization problem that asks for the shortest possible route that visits a given set of locations exactly once and then returns to the starting point. However, we can turn it into a decision problem by asking whether there exists a route with total distance less than a given value k. This decision problem can then be classified using NP-Completeness results. Similarly, the shortest path problem can be transformed into a decision problem by asking whether there exists a path between two nodes with a total distance less than k.

If this proof is successful, the optimisation problem is said to be NP-Hard, which means *at least as hard as the hardest NP problems*.

To prove that the longest simple cycle problem is NP-Complete, we can use a reduction from the Hamiltonian cycle problem. Given an instance of the Hamiltonian cycle problem, we can create an equivalent instance of the longest simple cycle problem by setting k to be the number of nodes in the graph. If the Hamiltonian cycle problem has a solution (a cycle that visits all nodes in the graph), then the longest simple cycle problem will have a solution (a cycle that visits at least k nodes). On the other hand, if the Hamiltonian cycle problem does not have a solution, then the longest simple cycle problem will not have a solution (because there is no cycle that visits all nodes). Thus, the longest simple cycle problem is NP-Complete.

To prove that the traveling salesman problem is NP-Complete, we can use a reduction from the Hamiltonian cycle problem. Given an instance of the Hamiltonian cycle problem, we can create an equivalent instance of the traveling salesman problem by setting the distance between each pair of nodes to be 1. If the Hamiltonian cycle problem has a solution (a cycle that visits all nodes in the graph), then the traveling salesman problem will have a solution (a route that visits all nodes and has a total distance equal to the number of nodes in the graph). On the other hand, if the Hamiltonian cycle problem does not have a solution, then the traveling salesman problem will not have a solution (because there is no route that visits all nodes). Thus, the traveling salesman problem is NP-Complete.

Bearing this in mind - have a go at proving maximum clique is NP-Hard

To prove that the maximum clique problem is NP-hard, we can use the fact that the clique decision problem (which asks whether a given graph has a clique of size k or greater) is NP-complete. Given a graph G and a value k, the clique decision problem can be solved in polynomial time by simply checking all possible subsets of the vertices in the graph and determining whether any of them form a clique of size k or greater.

To prove that the maximum clique problem is NP-hard, we can show that it is at least as difficult as the clique decision problem. In other words, we can demonstrate that an efficient solution to the maximum clique problem would also allow us to solve the clique decision problem in polynomial time.

To do this, we can start by assuming that we have an efficient algorithm for solving the maximum clique problem. Given a graph G, we can use this algorithm to find the size of the largest clique in the graph. Let this size be k. We can then use the clique decision problem to determine whether there is a clique of size k or greater in the graph. If the clique decision problem returns "yes", then we know that the maximum clique problem has a solution, and we can use the algorithm to find it. If the clique decision problem returns "no", then we know that the maximum clique problem does not have a solution.

Since the clique decision problem is NP-complete, this demonstrates that the maximum clique problem is NP-hard, because we have shown that an efficient solution to the maximum clique problem would allow us to solve the clique decision problem in polynomial time.

Polynomial Time Algorithms

Polynomial time algorithms are algorithms that can solve a problem in a time complexity of O(n^k) for some constant k, where n is the size of the input. These algorithms are considered efficient

because the time it takes to solve the problem grows at a polynomial rate as the size of the input increases. In contrast, algorithms with a time complexity of O(n!) or O(2^n) are considered inefficient because the time it takes to solve the problem grows exponentially as the size of the input increases. Many important problems in computer science, such as sorting and searching, can be solved using polynomial time algorithms.

So when faced with a novel problem, we now have two options:

1. Try to solve it in polynomial time
2. Try to prove it is NP-Complete, or NP-Hard in order to avoid wasting time trying to solve it exactly (we'll deal with what to do instead later…)

It is important to note that even if a problem is NP-Hard or NP-Complete, it does not mean that it is not possible to find efficient (though not necessarily polynomial time) algorithms to solve it. Many NP-Hard problems can be approximately solved in polynomial time using heuristics or approximation algorithms. These algorithms may not always find the optimal solution, but they can often find solutions that are close to optimal in a reasonable amount of time.

We've talked a little about the second option but could do with looking at the most common techniques for the first.

There are a few general approaches that are commonly used to design polynomial time algorithms:

1. Brute force: This involves trying all possible solutions and returning the best one. This can be efficient if the size of the input is small, but it is not suitable for larger inputs as the running time grows exponentially with the size of the input.
2. Divide and conquer: This approach involves dividing the problem into smaller subproblems and solving them individually, before combining the solutions to obtain the final answer. This can be efficient if the subproblems can be solved efficiently and the overall problem can be divided into a small number of subproblems.
3. Dynamic programming: This technique involves breaking the problem down into smaller subproblems and solving them in a specific order, storing the solutions to the subproblems in a table or array as they are computed. This allows for faster lookup of previously solved subproblems, which can be combined to solve larger subproblems and eventually the original problem.
4. Greedy algorithm: This approach involves making a series of locally optimal choices, with the hope that these choices will lead to a globally optimal solution. This can be efficient if the problem exhibits certain properties that allow for the locally optimal choices to lead to a globally optimal solution.
5. Linear programming: This technique involves formulating the problem as a set of linear constraints and then using a specific algorithm to find the solution that maximizes or minimizes a linear objective function subject to these constraints. This can be efficient if the problem can be expressed as a linear program.

After all, is you're struggling to prove your brand new problem is NP-Hard, it might mean that you can solve it efficiently!

There are several approaches that can be taken to try to solve a problem in polynomial time. One common approach is to try to find a pattern or structure in the problem that can be exploited to solve it efficiently. For example, if the problem involves searching for a particular element in a large dataset, a common approach is to use a data structure like a hash table or a binary search tree, which can perform the search in logarithmic time.

Another approach is to use dynamic programming, which involves breaking the problem down into smaller subproblems that can be solved independently and then combining the solutions to solve the larger problem. This can be an effective approach for problems that have an inherent recursive structure or that can be divided into smaller subproblems that can be solved independently.

A third approach is to use approximation algorithms, which are algorithms that aim to find a solution that is close to the optimal solution, rather than the optimal solution itself. These algorithms can be very useful in cases where it is difficult or impossible to find the optimal solution in polynomial time, but it is still possible to find a good approximation to the solution.

Finally, it is also possible to use heuristics, which are methods that use some form of trial and error to find a solution to a problem. Heuristics are often used to solve problems for which it is difficult to find an efficient algorithm, and they can be very effective in finding good solutions quickly.

Divide and Conquer

Divide and conquer is a common algorithmic technique used to solve problems in polynomial time. It involves dividing the problem into smaller subproblems, solving each of these subproblems individually, and then combining the solutions to the subproblems to get the solution to the original problem.

To use the divide and conquer approach, the problem must be able to be divided into smaller subproblems that are similar to the original problem in some way. These subproblems should be independent of each other, meaning that solving one subproblem should not affect the solution to any other subproblem.

The divide and conquer approach has several key steps:

1. Divide the original problem into smaller subproblems.
2. Solve each of the subproblems individually.
3. Combine the solutions to the subproblems to get the solution to the original problem.

One of the main advantages of the divide and conquer approach is that it allows us to solve problems more efficiently by taking advantage of parallelism. By solving the subproblems in parallel, we can potentially reduce the overall runtime of the algorithm.

Examples of problems that can be solved using the divide and conquer approach include the merge sort and quick sort algorithms for sorting lists, and the binary search algorithm for searching for an element in a sorted list.

Hopefully we are familiar with this technique:

Binary Search

Binary search is an algorithm that allows you to search for an element in a sorted list by dividing the list into two halves at each iteration. It works as follows:

1. Set the left index (L) to the first element of the list and the right index (R) to the last element of the list.
2. Check if the element at the middle index of the list (which is calculated by taking the average of L and R) is the element you are searching for. If it is, return the index.
3. If the element at the middle index is greater than the element you are searching for, set R to the middle index - 1 and go back to step 2.
4. If the element at the middle index is less than the element you are searching for, set L to the middle index + 1 and go back to step 2.

Binary search has a time complexity of O(log n), which makes it much faster than linear search (which has a time complexity of O(n)). However, it can only be used on sorted lists.

Mergesort

Mergesort is an efficient, general-purpose sorting algorithm that uses a divide and conquer approach to sort a given list of items. It works by dividing the list into two halves, sorting each half, and then merging the sorted halves back together.

The key to the efficiency of mergesort is the merge operation, which combines two sorted lists into a single, sorted list in linear time. This is achieved by maintaining two pointers, one for each list, and continually comparing the elements at the pointers to determine which should be added to the final sorted list.

To sort a list using mergesort, the following steps are taken:

1. If the list has 0 or 1 elements, it is already sorted, so return it.
2. If the list has more than 1 element, divide it into two halves.
3. Sort the two halves using mergesort.
4. Merge the two sorted halves back together using the merge operation.

The time complexity of mergesort is O(n log n), making it more efficient than many other sorting algorithms, especially for large lists. It is also a stable sort, meaning that it preserves the relative order of items with equal keys.

Quicksort

Quicksort is a divide and conquer algorithm that is used to sort an array or list of elements. It works by selecting a "pivot" element from the list and partitioning the other elements into two sub-lists, according to whether they are less than or greater than the pivot. The sub-lists are then recursively sorted using the same process, until the entire list is sorted.

Quicksort has an average case time complexity of O(n log n), making it more efficient than many other sorting algorithms, such as bubble sort and insertion sort. It is also relatively easy to implement and can be done in-place (meaning it doesn't require additional memory). However, its worst case time complexity is O(n^2), which can occur if the pivot is consistently chosen poorly.

Therefore, it is important to choose a pivot that is roughly the median of the list in order to avoid the worst case scenario.

Reduction (parallel programming pattern)

In parallel programming, reduction is a common pattern that involves combining the results of multiple computations in a parallel program into a single result. The idea is to divide the problem into smaller subproblems, solve each of these subproblems in parallel, and then combine the results of these subproblems to obtain the final result.

One way to implement reduction is to use a tree-based approach, where the subproblems are solved in a tree-like structure and the results are combined as they move up the tree. This can be done by assigning each subproblem to a separate processor or thread, and having these processors or threads communicate with each other as they combine their results.

Another way to implement reduction is to use a pipeline approach, where the subproblems are solved in a linear fashion and the results are combined as they move through the pipeline. This can be done by assigning each subproblem to a separate stage in the pipeline and having these stages communicate with each other as they pass their results along.

Reduction can be an effective way to solve problems in parallel because it allows us to take advantage of the parallelism available in the system, and it can be implemented using a wide range of algorithms and programming languages.

Greedy Algorithms

A greedy algorithm is an algorithmic paradigm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum. In other words, a greedy algorithm makes the most locally beneficial choice at each step and hopes that these choices lead to a globally optimal solution.

Here is a simple example of a greedy algorithm:

Imagine you are a cashier and you need to give a customer their change using the least amount of coins possible. A greedy approach would be to give the customer the largest denominations of coins first, as this would minimize the number of coins needed. For example, if the customer needs to receive 67 cents in change, a greedy algorithm would give them two quarters (50 cents), one dime (10 cents) and two pennies (2 cents) instead of six nickels (30 cents).

While greedy algorithms can be effective in finding a solution, they are not always the most efficient and can sometimes lead to suboptimal solutions. It is important to carefully evaluate the problem before deciding to use a greedy algorithm.

Greedy algorithms involve constructing a solution by selecting the best option at each step.

For example, the problem of finding the minimum number of coins to make a given amount of money can be solved using a greedy algorithm. At each step, the algorithm selects the largest available coin that does not take the total amount of money over the target amount, until the target amount is reached.

This approach is called greedy because it makes a locally optimal choice at each step in the hope of finding a global optimum. However, not all problems can be solved using a greedy approach and it is important to carefully consider whether it is suitable for a particular problem.

One way to prove that a greedy algorithm is correct is to use the greedy choice property, which states that if the algorithm makes a locally optimal choice at each step, it will find a globally optimal solution. It is important to note that the greedy choice property does not always hold and it is necessary to carefully prove that it holds for a particular problem before using a greedy algorithm to solve it.

A simple example is finding the minimum number of coins to give as change.

At each step, we choose the largest coin that does not exceed the remaining amount of change that we need to give. This is an optimal strategy because we want to minimize the number of coins used, and using a larger coin will always be better than using multiple smaller coins to add up to the same value.

Another example of a greedy algorithm is the Huffman coding algorithm for data compression. In this algorithm, we construct a prefix code by repeatedly merging the two symbols with the smallest frequencies and assigning a 0 or 1 to each branch of the resulting tree. This is optimal because we want to minimize the length of the encoded message, and merging symbols with the smallest frequencies will always result in a shorter code than merging symbols with larger frequencies.

There are many other examples of greedy algorithms, including the Dijkstra's algorithm for finding the shortest path in a graph and the Kruskal's algorithm for finding a minimum spanning tree.

Or Prim’s algorithm for the minimum spanning tree problem

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. It starts with a random vertex and adds the minimum weight edge to it that connects it to a vertex not yet in the tree. It repeats this process until all vertices are in the tree. The resulting tree is a minimum spanning tree because at each step, the algorithm chooses the minimum weight edge that connects the tree to a vertex not in the tree, so the total weight of the edges in the tree is minimized.

Here is an example of Prim's algorithm implemented in Python:

```
from typing import List, Tuple

def find_minimum_spanning_tree(n: int, edges: List[Tuple[int, int, int]]) -> int:
    # Initialize the minimum spanning tree to be empty
    mst = []
    # Create a set to store the vertices that have been visited
    visited = set()
    # Sort the edges by weight
    edges.sort(key=lambda x: x[2])

    # Select a starting vertex
    current_vertex = 0
    visited.add(current_vertex)

    while len(visited) < n:
```

```
        # Find all the edges that connect the current vertex to unvisited vertices
        unvisited_edges = [e for e in edges if e[0] == current_vertex and e[1] not in visited or e[1] ==
current_vertex and e[0] not in visited]
        # Select the edge with the minimum weight
        next_edge = min(unvisited_edges, key=lambda x: x[2])
        # Add the edge to the minimum spanning tree
        mst.append(next_edge)
        # Add the new vertex to the set of visited vertices
        visited.add(next_edge[0] if next_edge[1] == current_vertex else next_edge[1])
        # Set the current vertex to be the new vertex
        current_vertex = next_edge[0] if next_edge[1] == current_vertex else next_edge[1]

    # Return the total weight of the minimum spanning tree
    return sum(e[2] for e in mst)
```

This implementation of Prim's algorithm has a time complexity of O(nlogn) because it sorts the edges at the beginning of the algorithm, which takes O(nlogn) time. The rest of the algorithm consists of simple operations that take constant time, so the overall time complexity is O(nlogn).

Even better, the algorithm for generating Huffman Codes is greedy

Here is an example of a python implementation of Huffman encoding

```
import heapq
from collections import defaultdict

def huffman_encoding(data):
    # Create a frequency table for the input data
    freq_table = defaultdict(int)
    for ch in data:
        freq_table[ch] += 1

    # Use the frequency table to create a priority queue of tuples (frequency, character)
    pq = [(freq, ch) for ch, freq in freq_table.items()]
    heapq.heapify(pq)

    # While there is more than one element in the priority queue:
    while len(pq) > 1:
        # Pop the two elements with the lowest frequency
        freq1, ch1 = heapq.heappop(pq)
        freq2, ch2 = heapq.heappop(pq)

        # Create a new tuple representing a binary tree with ch1 and ch2 as its children
        # and with frequency equal to the sum of the frequencies of ch1 and ch2
        tree = (freq1 + freq2, ch1, ch2)

        # Add the new tree back into the priority queue
        heapq.heappush(pq, tree)

    # The remaining element in the queue is the root of the binary tree, which represents
    # the whole encoded message. Recursively traverse the tree to generate the encoding.
    encoding = {}
    def generate_encoding(tree, prefix=''):
        if len(tree) == 2:  # tree is a leaf (character)
            encoding[tree[1]] = prefix
        else:  # tree is a binary tree with two children (subtrees)
            generate_encoding(tree[1], prefix + '0')
            generate_encoding(tree[2], prefix + '1')
```

```
    generate_encoding(pq[0])

    # Encode the input data using the generated encoding
    encoded_data = ''
    for ch in data:
        encoded_data += encoding[ch]

    return encoded_data, encoding

# Test the huffman_encoding function
encoded_data, encoding = huffman_encoding('this is a test')
print(encoded_data)
print(encoding)
```

This implementation first creates a frequency table for the input data, which counts how many times each character appears. It then uses the frequency table to create a priority queue of tuples (frequency, character), where the frequency is used to order the elements in the queue.

Next, the algorithm repeatedly pops the two elements with the lowest frequency from the queue and combines them into a new tuple representing a binary tree with the two characters as its children and with a frequency equal to the sum of the frequencies of the two characters. This process continues until there is only one element left in the queue, which is the root node of the Huffman tree.

Finally, the algorithm generates the Huffman codes by traversing the tree and assigning a code of 0 to the left child of each node and a code of 1 to the right child. The resulting codes are stored in a dictionary, where the keys are the characters and the values are the codes.

Here is a simple example of how to use the Huffman coding algorithm in Python:

```
from collections import Counter
from heapq import heappush, heappop

def huffman_codes(data):
    # Create a frequency table for the input data
    frequency_table = Counter(data)

    # Create a priority queue of tuples (frequency, character)
    heap = [(freq, char) for char, freq in frequency_table.items()]
    heapify(heap)

    # Repeatedly combine the two elements with the lowest frequency
    while len(heap) > 1:
        freq1, char1 = heappop(heap)
        freq2, char2 = heappop(heap)
        heappush(heap, (freq1 + freq2, (char1, char2)))

    # Generate the Huffman codes by traversing the tree
    codes = {}
    def generate_codes(node, code):
        if isinstance(node, str):
            codes[node] = code
        else:
            generate_codes(node[0], code + "0")
            generate_codes(node[1], code + "1")
```

```
    generate_codes(heap[0][1], "")

    return codes

# Test the algorithm with a simple example
data = "aaabbc"
codes = huffman_codes(data)
print(codes)  # {'a': '00', 'b': '01', 'c': '1'}
```

Dynamic Programming

Dynamic programming is an algorithm design technique that is used to solve optimization problems by breaking them down into smaller subproblems and storing the solutions to these subproblems in a table. It is called dynamic programming because it involves making a sequence of decisions, each of which depends on the decisions made before it.

The key idea behind dynamic programming is that the optimal solution to a problem can be obtained by solving its subproblems and combining their solutions in a specific way. To solve a problem using dynamic programming, we typically follow these steps:

Identify the subproblems: Identify the subproblems that need to be solved in order to find the solution to the original problem.

Store the solutions to the subproblems: Create a table or array to store the solutions to the subproblems as they are solved.

Solve the subproblems: Solve the subproblems in a specific order, typically starting with the smallest subproblems and working up to the larger ones.

Combine the solutions to the subproblems: Use the solutions to the subproblems to find the solution to the original problem.

Dynamic programming is typically used to solve optimization problems, such as finding the shortest path between two points, or the least number of coins needed to make a certain amount of change. It is particularly useful for problems that can be divided into overlapping subproblems, such as the Fibonacci sequence or the knapsack problem.

Dynamic programming is similar to divide and conquer in the sense that it solves a larger problem by first solving a series of smaller problems and then remembering their solutions.

However, dynamic programming differs in that it typically involves using these solutions to construct a solution to the original problem, rather than simply combining the solutions of the smaller subproblems.

Dynamic programming algorithms are usually used to solve optimization problems, where the goal is to find the best possible solution among a set of feasible solutions. They work by building up a solution to the problem from smaller subproblems, rather than solving the problem directly.

For example, the Knapsack problem can be solved using dynamic programming. In this problem, we are given a knapsack with a certain capacity, and a set of items with different weights and values. The goal is to fill the knapsack with items in such a way that we maximize the total value of the items while staying within the capacity of the knapsack.

To solve this problem using dynamic programming, we first create a two-dimensional array to store the solutions to the subproblems. We then fill in the array by considering each item in turn, and deciding whether or not to include it in the knapsack based on its weight and value. By storing the solutions to the subproblems in the array, we can avoid having to recalculate them, which allows us to solve the problem more efficiently.

However, the difference is that in dynamic programming, unlike in divide and conquer, the subproblems may overlap.

This means that the same subproblem may be solved multiple times, so dynamic programming algorithms use a table or an array to store the solutions to the subproblems, which can be looked up when needed. This is called "memoization."

One classic example of a problem that can be solved using dynamic programming is the Fibonacci sequence problem, where the goal is to find the nth number in the Fibonacci sequence. This problem can be solved using a recursive approach, but it is inefficient because many of the subproblems are solved multiple times. By using dynamic programming and storing the solutions to the subproblems in an array, we can avoid the unnecessary recomputation and solve the problem in a more efficient way.

Another common example is the Knapsack problem, where the goal is to maximize the value of items that can be placed in a knapsack with a given weight capacity. This problem can also be solved using dynamic programming by creating a table of the maximum value that can be obtained for each weight capacity and each subset of items.

We solve each subproblem once and try to reuse the solutions in order to reduce our search from exponential to polynomial time. This gives us a powerful technique.

Dynamic programming is often used to solve optimization problems, where the goal is to find the optimal solution (either the maximum or minimum value) among all possible solutions.

To solve a problem using dynamic programming, we need to follow these steps:

1. Identify the subproblems: Divide the original problem into smaller subproblems.
2. Recursively solve the subproblems: Solve each subproblem recursively, and store the solutions in a table or array.
3. Combine the solutions: Use the solutions of the subproblems to construct the solution for the original problem.
4. Optimize the solution: Use techniques such as memoization to avoid solving the same subproblem multiple times and improve the efficiency of the algorithm.

An example of a problem that can be solved using dynamic programming is the Knapsack problem, which involves selecting a set of items with maximum value, subject to a constraint on the total weight of the items.

Typesetting

Typesetting refers to the process of arranging and formatting text for printing or display. It involves selecting the appropriate font, size, and layout for the text, and may also include features such as

hyphenation, kerning, and justification. Typesetting is usually done using specialized software, such as Adobe InDesign or LaTeX, which allows for precise control over the appearance and layout of the text. Typesetting is often used in the publishing industry, but it can also be applied to other fields, such as advertising, marketing, and web design.

Consider the problem whereby we have *n* words we wish to fit on a page which is *k* characters wide. We want to justify the text whilst minimising the size of the gaps between words.

Typesetting is the process of formatting text and images for publication, such as in a book or newspaper. The goal of typesetting is to create a visually appealing and easy-to-read layout for the text. In order to do this, the typesetter must consider factors such as font size, font type, line spacing, and margins. One specific problem in typesetting is the justification of text, which refers to aligning the text along both the left and right margins of a page. This can be a challenging problem, especially when trying to minimize the size of the gaps between words. One way to approach this problem is through the use of dynamic programming, by breaking the text down into smaller chunks and solving the justification problem for each chunk individually before combining the solutions to form the final layout.

- Greedy algs/divide and conquer don't work so well.

  Dynamic programming is a good approach for this problem. We can start by trying to fill the first line with as many words as possible, then use the remaining space to determine the size of the gaps between the words. We can then use this information to fill the next line, and so on, until we have filled the entire page. By keeping track of the minimum gap size at each step, we can ensure that we are always making the optimal decision.

- Dynamic programming does:

  We start by considering the problem of typesetting the first i words of the text. Let g(i) be the optimal solution for this problem. We can then find the value of g(i) by considering the value of g(i-1) and the size of the gaps we would need to insert in order to typeset the ith word. If we let w(i) be the length of the ith word and S(i) be the sum of the lengths of the first i words, then we can compute g(i) as follows:

  g(i) = min(g(i-1) + 2 * (k - S(i)))

  This algorithm has a time complexity of O(nk), which is polynomial in the size of the input.

    - If we guess how many words are on the first line and solve the remaining lines by recursion, we find the subproblems repeat

      - So, we can store the solutions to each subproblem in a table and reuse them rather than recomputing them.
      - To do this, we first create a table of size nxk where n is the number of words and k is the number of characters per line.

- Then, we iterate through the table, filling in the optimal solution for each subproblem.
- For each cell in the table, we consider all possible ways of breaking the text such that the line ends at that cell.
- We calculate the cost of each break by summing the squares of the sizes of the gaps between the words and selecting the minimum cost.
- Finally, we return the optimal solution for the last cell in the table, which represents the optimal solution for the entire problem.

- In fact, there are only *n* subproblems in total, so once we solve any one of them, we enter the answer into a lookup table

  and then whenever we need to solve it again, we simply look it up in the table. This is much faster than recomputing it.

  To implement this in code, we can use a table to store the minimum cost of typesetting the words from i to j, where i and j are indices into the list of words. Then, we can fill in the table using the following recursive formula:

  min_cost[i][j] = min(min_cost[i][j], min_cost[i][k] + min_cost[k+1][j])

This formula states that the minimum cost of typesetting the words from i to j is the minimum of the cost of typesetting them using a line break after every kth word, for all values of k from i to j-1.

To compute the final cost, we simply look up the value in min_cost[0][n-1], where n is the number of words.

We can also use memoization to avoid recomputing subproblems, or use a bottom-up approach to fill in the table in a more efficient manner.

Shortest paths

The shortest path problem is a problem of finding a path between two vertices (or nodes) in a graph such that the total sum of the weights of its edges is minimized. This problem can be solved using various algorithms such as Dijkstra's algorithm, Bellman-Ford algorithm, or Floyd-Warshall algorithm. These algorithms work by starting at the source vertex and relaxing the edges in the graph, gradually discovering the shortest path to the destination vertex.

In fact, dynamic programming can be seen as a means of solving a huge range of problems by transforming them into an instance of *shortest paths*, which can then be solved using the Bellman-Ford or Dijkstra algorithms.

Some examples of problems that can be solved using dynamic programming include:

- Knapsack problem: given a set of items, each with a weight and a value, determine the subset of items that can be included in a bag with a maximum weight capacity such that the total value is maximized.

- Longest common subsequence: given two strings, find the longest subsequence that is present in both strings.
- Matrix chain multiplication: given a chain of matrices, determine the optimal way to multiply them together in order to minimize the number of scalar multiplications required.
- Fibonacci numbers: given a number n, compute the nth Fibonacci number using a recursive algorithm that stores previously computed values in a lookup table.

To solve these problems using dynamic programming, we typically start by defining a recurrence relation that describes the relationship between subproblems and then use this recurrence relation to build a table of values that can be used to compute the solution to the original problem.

- Normally Bellman-Ford, which is essentially the recursive method shown.

  The Bellman-Ford algorithm is a method for finding the shortest paths between nodes in a graph, where the graph may contain negative-weight edges. It works by iteratively relaxing the shortest path estimates of each node until they converge to the true shortest paths. The algorithm maintains a distance table, with one row for each node in the graph, and one column for each other node. Initially, the distance from each node to itself is zero, and the distance to all other nodes is set to infinity. The algorithm then iteratively updates the distance table according to the following rule: for each edge (u, v) in the graph, if the distance from u to v is greater than the distance from u to v plus the weight of the edge, then update the distance from u to v to be the distance from u to v plus the weight of the edge. The algorithm continues iterating until the distance table stops changing, at which point it is guaranteed to have found the shortest paths from the start node to all other nodes in the graph.

  Here is an example of how to use the Bellman-Ford algorithm to find the shortest path between two nodes in a graph:

```
from collections import defaultdict

def bellman_ford(graph, start, end):
 # Create a dictionary to store the distances from the start node to all other nodes
 distances = defaultdict(lambda: float('inf'))
 distances[start] = 0

 # Repeat the relaxation process |V|-1 times (since the shortest path can go through at most |V|-1
edges)
 for i in range(len(graph) - 1):
  # Iterate through all edges in the graph
  for u, v, weight in graph:
   # If going through this edge would give us a shorter distance to v, update the distance to v
   if distances[u] + weight < distances[v]:
    distances[v] = distances[u] + weight

 # Return the shortest distance from the start node to the end node
 return distances[end]

# Example graph
graph = [
 (0, 1, 3),
```

```
    (0, 2, 2),
    (1, 2
```

This code will output the shortest path from node 0 to node 3 in the graph as 2.

- If we make all identical subproblems into a single ‘node’ we transform the recursion tree into a directed graph.

Here is an example of using Dijkstra's algorithm to find the shortest path in a graph using Python:

```
import sys
import heapq

def dijkstra(graph, start, end):
  # initialize distances and previous nodes
  distances = {node: sys.maxsize for node in graph}
  distances[start] = 0
  previous = {node: None for node in graph}

  # create a min heap of nodes sorted by distance
  heap = []
  heapq.heappush(heap, (0, start))

  while heap:
    # get the node with the smallest distance
    distance, node = heapq.heappop(heap)

    # if we have reached the end node, we are done
    if node == end:
      break

    # update distances for all neighbors
    for neighbor, cost in graph[node].items():
      new_distance = distance + cost
      if new_distance < distances[neighbor]:
        distances[neighbor] = new_distance
        previous[neighbor] = node
        heapq.heappush(heap, (new_distance, neighbor))

  # reconstruct the shortest path
  path = []
  node = end
  while node is not None:
    path.append(node)
    node = previous[node]

  # return the reversed path
  return path[::-1]

# example graph
graph = {
  'A': {'B': 3, 'C': 5, 'D': 8},
  'B': {'C': 1, 'D': 2},
```

```
  'C': {'D': 4},
  'D': {}
}

shortest_path = dijkstra(graph, 'A', 'D')
print(shortest_path)  # ['A', 'B', 'C', 'D']
```

Here is an example of the Bellman-Ford algorithm implemented in Python, with detailed line-by-line comments:

```
# import sys module for sys.maxsize
import sys

# define Graph class
class Graph:

    # Constructor function to initialize the graph
    def __init__(self, vertices):
        self.V = vertices  # number of vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    # function to add an edge to the graph
    def add_edge(self, u, v, w):
        self.graph[u][v] = w

    # utility function used to print the solution
    def print_solution(self, dist):
        print("Vertex Distance from Source")
        for node in range(self.V):
            print(node, "t", dist[node])

    # The main function that finds the shortest distance from src to
    # all other vertices using Bellman-Ford algorithm. The function
    # also detects negative weight cycle
    def bellman_ford(self, src):
        # step 1: initialize distance from src to all other vertices as infinite
        dist = [sys.maxsize] * self.V
        dist[src] = 0

        # step 2: relax edges |V| - 1 times
        for i in range(self.V - 1):
            # iterate through all edges
            for u in range(self.V):
                for v in range(self.V):
                    # step 3: check if relaxation is possible
                    if dist[u] != sys.maxsize and dist[u] + self.graph[u][v] < dist[v]:
                        dist[v] = dist[u] + self.graph[u][v]

        # step 4: check for negative-weight cycles. The above step
        # guarantees shortest distances if graph doesn't contain
        # negative weight cycle. If we get a shorter path, then there
        # is a cycle.
```

```
    for u in range(self.V):
        for v in range(self.V):
            if dist[u] != sys.maxsize and dist[u] + self.graph[u][v] < dist[v]:
                print("Graph contains negative weight cycle")
                return

    # if we reach this point, then the graph doesn't contain
    # negative weight cycle and we print the solution
    self.print_solution(dist)
```

This implementation of the Bellman-Ford algorithm finds the shortest distance from the source vertex to all other vertices in a weighted directed graph. It works by relaxing the edges of the graph |V| - 1 times, where |V| is the number of vertices in the graph. In each iteration, the algorithm checks if the distance to the destination vertex can be shortened by going through an intermediate vertex. If so, the distance is updated and the predecessor vertex is recorded. After the |V| - 1 iterations, the algorithm checks for negative weight cycles (which would mean that the distance can be further shortened) and returns the distances and predecessors if no such cycles are found.

Here is an example of the algorithm in Python with detailed line-by-line comments:

```
def bellman_ford(graph, source):
    # Initialize distances and predecessors
    distances = [float('inf') for _ in range(len(graph))]
    predecessors = [None for _ in range(len(graph))]
    distances[source] = 0

    # Relax edges |V| - 1 times
    for _ in range(len(graph) - 1):
        for u, v, w in graph.edges():
            # Update distance if going through u gives a shorter path to v
            if distances[u] + w < distances[v]:
                distances[v] = distances[u] + w
                predecessors[v] = u

    # Check for negative weight cycles
    for u, v, w in graph.edges():
        if distances[u] + w < distances[v]:
            print("Graph contains a negative weight cycle")
            return None, None

    return distances, predecessors
```

- We can then use the objective function to define weights on the edges of the graph.

The objective function is the function we are trying to optimize, in this case the size of the gaps between words. By defining weights on the edges as the cost of transitioning from one subproblem to another, we can use shortest path.

Linear and Integer Programming

Linear programming is a mathematical method used to find the maximum or minimum value of a linear objective function, subject to a set of linear inequality or equality constraints. It can be used to optimize allocation of limited resources and is commonly used in business and economics.

Integer programming is a variant of linear programming where the variables are required to be integers rather than continuous values. This makes integer programming more difficult to solve than linear programming, as it is not always possible to find integer solutions for certain problems. However, integer programming is useful for modeling problems where only integer values are allowed, such as the number of items that can be produced or the number of employees that can be hired.

Both linear programming and integer programming can be solved using a variety of algorithms, including the simplex algorithm and branch and bound.

Linear programming

We left out one very important polynomial time technique last week as it deserves a class of its own. That is *linear programming*, the process of representing a problem using a certain form and then using a standard approach (usually a *solver*) to tackle it.

Linear programming involves optimizing a linear objective function subject to a set of linear inequality or equality constraints. It can be used to solve problems such as finding the minimum cost to produce a certain number of products or finding the maximum profit given certain constraints.

Integer programming is a variation of linear programming where some or all of the variables must be integers. This can be used to solve problems such as finding the minimum number of coins to give as change or scheduling problems where resources must be assigned in whole units.

Both linear and integer programming can be solved using algorithms such as the simplex algorithm or the branch and bound method. These algorithms are efficient and widely used in practice to solve a wide variety of optimization problems.

This technique is one of the earlier forms of combinatorial optimisation and involves minimising (or maximising) a linear function, subject to a number of linear constraints.

Integer programming is a generalization of linear programming in which some or all of the variables must be integers. This is often used to model problems where the values of the variables have to be whole numbers, such as in the case of counting problems. The algorithms used to solve integer programming problems are similar to those used for linear programming, but they are generally slower and more complex.

The notion of a *linear* function, means that the function must draw straight lines (or planes/hyperplanes) when drawn on a graph. This is always the case as long as we do not multiply our variables by one another (or by themselves)

Linear programming is a powerful tool and is used in a wide range of applications including resource allocation, scheduling and manufacturing. It is also a basis for integer programming, which involves finding an optimal solution using only integer values, rather than continuous values. Integer

programming is useful in situations where the variables must be integers, such as when the variables represent discrete quantities.

A simple example

Maximise 3x + 4y - z

Subject to x + y <= 12

z >= 3

This problem is asking us to find the maximum value of the function 3x + 4y - z, while ensuring that the values of x and y do not exceed 12 when added together, and that the value of z is at least 3.

To solve this problem, we can use a linear programming solver, which will find the optimal solution for us. In this case, the optimal solution would be the values of x, y, and z that give us the maximum value of 3x + 4y - z while still satisfying the constraints.

The function to be maximised (or minimised) is called the *objective function* and we want to find the largest (or smallest) value for it which satisfies all the constraints.

Linear programming problems are typically solved using the simplex algorithm, which is a polynomial time algorithm for finding the optimal solution. Integer programming is a variant of linear programming where the variables are required to take on integer values. This can be more difficult to solve as it requires more computation, but there are specialised algorithms for solving integer programming problems efficiently.

A little geometry and the simplex algorithm

The solution to a linear programming problem is always at one of the vertices of the feasible region, which is the region defined by all the constraints. This is because if we are at any other point within the region, we can always move to a vertex which would increase the value of the objective function.

The simplex algorithm is a method for finding the optimal vertex of the feasible region, which is either the vertex with the maximum value of the objective function (in the case of maximization) or the vertex with the minimum value of the objective function (in the case of minimization). The algorithm works by starting at a vertex and moving to a neighbouring vertex that improves the value of the objective function, until it reaches an optimal vertex.

The advantage of representing a problem using linear functions + constraints is that the optimum (maximum or minimum) solution is always

at a corner (or vertex) of the feasible region - that is, the region which satisfies all the constraints. This is because the objective function is linear and so its graph will always be a straight line.

- On the surface of the shape formed by the constraints (called the ‘feasible region’)

  This means that we only need to check the values at the vertices, which will be a polynomial number of them. The simplex algorithm does this.

Note that the feasible region will always be a convex shape, as the constraints are themselves linear.

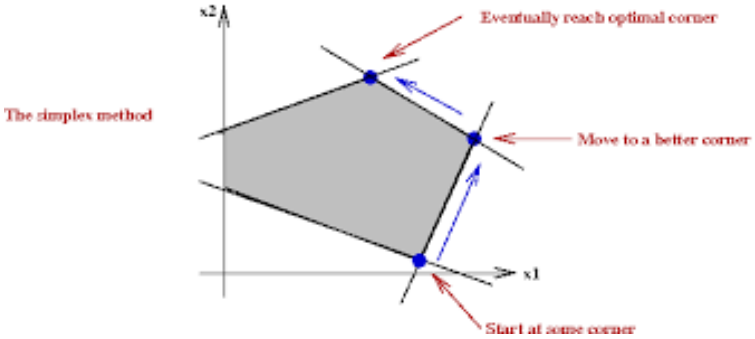
A variant of this technique is integer programming, which is the same as linear programming but with the added constraint that the variables must be integers. This can be much harder to solve and there is no general polynomial time solution.

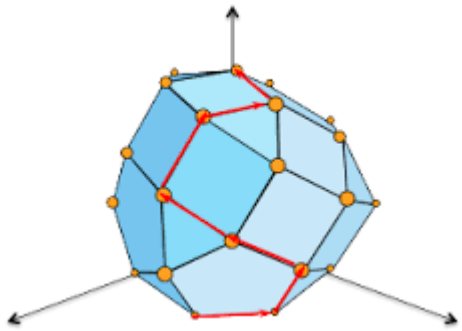
- Can be found at one of the corners
    - of the feasible region (this is always the case because linear functions can only ever draw straight lines).
    - This means that we only have to check the vertices of the feasible region in order to find the optimum solution. This is the idea behind the simplex algorithm, which solves linear programming problems by iteratively moving from one vertex of the feasible region to another, always improving the value of the objective function.
- Reachable by a simple 'hill climbing' or 'iterative improvement' technique

    The simplex algorithm is an example of an iterative improvement algorithm that is used to solve linear programming problems. It works by starting at a corner of the feasible region and iteratively improving the solution until it reaches an optimal one. It does this by finding the next corner that results in a better solution and repeating the process until no further improvement is possible. The algorithm terminates when it reaches a corner that is a local maximum or minimum, which is also the global maximum or minimum for the objective function within the feasible region.

    - Start at a corner and keep moving to a neighbouring corner with better objective value

This is the basis of the simplex algorithm, which is a popular method for solving linear programming problems. The algorithm starts at a corner of the feasible region and iteratively moves to a nearby corner that improves the objective function. This process is repeated until the optimal solution is reached.





Simplex algorithm

The simplex algorithm is of some theoretical interest as it can be used to solve linear programming problems in polynomial time, which was a major breakthrough when it was first developed. It is also widely used in practice, as it is relatively easy to implement and can solve large problems efficiently.

Linear programming has many practical applications, including finding the optimal allocation of resources, such as time and money, in business and logistics problems. It is also used in economics, engineering and many other fields.

Integer programming is a generalization of linear programming, where some or all of the variables are restricted to be integers. These problems are generally more difficult to solve than linear programming problems, as the feasible region is no longer a continuous space but rather a set of discrete points. However, integer programming can still be solved in polynomial time using specialized algorithms.

- It has been shown to run in exponential time in the worst case whilst still being very efficient in practice.

- It is also used in integer programming, which is a generalization of linear programming where the variables are required to be integers rather than continuous. This can be useful for problems that require discrete solutions, such as scheduling or resource allocation.
- ● Linear programming and integer programming are widely used in various fields such as economics, computer science, and operations research. They have many applications, including optimization of production and transportation, portfolio optimization in finance, and optimization of supply chain management.
- ● There are many different algorithms and solvers available for linear and integer programming, including the simplex algorithm, interior-point methods, branch and bound, and cutting plane methods. In practice, it is often best to use a solver rather than implementing the algorithm yourself as they are highly optimized and can handle large scale problems efficiently.
- There are polynomial time alternatives, called *interior point* methods (which cross the middle of the feasible region rather than the edges) but these often perform *less* efficiently than the simplex method in practice.

  Linear programming is just one type of optimization problem, another important one is integer programming, which is similar to linear programming but requires the variables to be integers rather than real numbers. Like linear programming, integer programming can be

used to represent and solve a wide range of optimization problems, including scheduling, resource allocation, and network design problems. The branch of mathematics that deals with the optimization of integer variables is called integer programming, and it is an important area of research in operations research and computer science.

- It is an iterative improvement technique which guarantees to find the optimum solution

  It works by starting at a corner of the feasible region and moving to a neighbouring corner which has a better objective value. This process is repeated until no such move is possible, at which point the algorithm has reached the optimal solution.

  - Because of the linear constraints, it never gets 'stuck' in suboptimality.

    It is quite easy to implement and is widely available in libraries and software packages.

    Here is a simple implementation of the simplex algorithm in Python:

```
def simplex(c, A, b, basic_vars):
 """
 Solves the linear programming problem:
 maximize    c^T x
 subject to  Ax <= b
             x >= 0
 using the simplex algorithm.
 Args:
  c: coefficients of the objective function
  A: constraints matrix
  b: constraints vector
  basic_vars: indices of the basic variables
 Returns:
  A tuple (optimal_value, optimal_solution).
 """
 n = len(c)  # number of variables
 m = len(b)  # number of constraints

 # initialize the tableau
 tableau = [-c] + [list(row) for row in A]
 tableau.append(b)
 tableau = [[tableau[i][j] if j < n else 0 for j in range(n+m+1)]
          for i in range(m+1)]
 for i in range(m):
  tableau[i][n+i] = 1
 tableau[m][n+m] = 1

 while True:
  # find the entering variable (minimum positive ratio)
  min_ratio = float("inf")
  entering_var = None
  for j in range(n):
   if tableau[-1][j] > 0:
    min_ratio = min(min_ratio, tableau[-1][-1]/tableau[-1][j])
    entering_var = j
  if min_ratio == float("inf"):
```

```
        # optimal solution found
        return tableau[-1][-1], [tableau[i][-1] for i in range(m)]

    # find the leaving variable (minimum positive ratio)
    min_ratio = float("inf")
    leaving_var = None
    for i in range(m):
        if tableau[i][entering_var] > 0:
            min_ratio = min(min_ratio, tableau[i][-1]/tableau[i][entering_var])
            leaving_var = i
    if min_ratio == float("inf"):
        # unbounded solution
        return float("inf"), []

    # pivot
    pivot = tableau[leaving_var][entering_var]
    tableau[leaving_var] = [x/pivot for x in tableau[leaving_var]]
    for i in range(m+1):
        if i != leaving_var:
            multiple = tableau[i][entering_var]
            tableau[i] = [tableau[i][j] - multiple*tableau[leaving_var][j]
                          for j in range(n+m+1)]
```

- We'll investigate this much more later, with problems that don't fit such a convenient shape.

This implementation of the simplex algorithm takes as input the coefficients of the objective function `c`, the constraints matrix `A`, the constraints vector `b`, and the indices of the basic variables `basic_vars`. It returns a tuple containing the optimal value and the optimal solution.

solution by finding a corner of the feasible region which gives an improved objective value and then updating the values of the variables to reflect this new corner. This process is repeated until no more improvement is possible.

To implement the simplex algorithm in Python, we can use the PuLP library. The following example shows how to use PuLP to solve the linear programming problem defined above:

```
# Import PuLP and create the problem object
import pulp
prob = pulp.LpProblem("Maximisation Problem", pulp.LpMaximize)

# Define the variables
x = pulp.LpVariable("x", lowBound=0)
y = pulp.LpVariable("y", lowBound=0)
z = pulp.LpVariable("z", lowBound=3)

# Define the objective function
prob += 3*x + 4*y - z

# Define the constraints
prob += x + y <= 12

# Solve the problem
status = prob.solve()
```

```
# Print the solution
print(f"Optimal solution: {pulp.value(prob.objective)}")
print(f"x = {x.varValue}")
print(f"y = {y.varValue}")
print(f"z = {z.varValue}")
```

This code will output the optimal solution, as well as the values of the variables at the optimal solution. Note that we have defined the variables x, y, and z with lower bounds of 0, 0, and 3 respectively, which corresponds to the inequality constraints in the problem definition.

Integer programming

Linear programming works excellently when we have a continuous objective function and constraints. It becomes less effective if any of the variables must take on certain values (e.g. must be integers or booleans)

In this case, we can use integer programming, which is similar to linear programming but with the added restriction that some or all of the variables must take on integer values. This makes the problem more difficult to solve as it is no longer possible to simply move from one corner of the feasible region to another, as the corners may not correspond to integer solutions. As a result, integer programming problems can be much harder to solve than their linear programming counterparts and may require more specialized algorithms and/or heuristics.

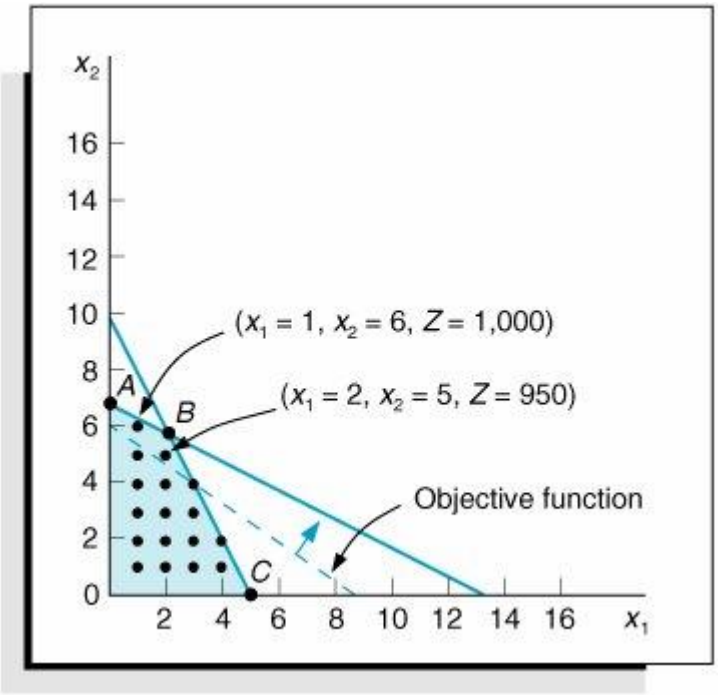
The difficulty with this is that it is much harder to find the optimal solution in this case, as we can no longer use the simplex algorithm. Instead, we must use integer programming techniques which involve solving a series of linear programming problems. These techniques are much slower in practice, although they still run in polynomial time.

- The optimum is not necessarily at a corner

  The optimum solution may not be a feasible solution, because it may not satisfy the integrality constraints

- It cannot always be found by rounding the optimum corner up or down

  This is where integer programming comes in. It is a variant of linear programming where we add the additional constraint that all variables must be integers. This is a much harder problem to solve and there is no guarantee that we can find the optimal solution in polynomial time, even though it is still in the class NP. In practice, integer programming solvers can be very effective but can sometimes take a long time to find a solution, especially for larger problems.



What does this mean?

We can represent instances of subset sum as an integer programming problem

This means that when we have variables that must take on certain values (e.g. must be integers or booleans), the optimal solution may not be at a corner of the feasible region and may not be found by simply rounding the optimal solution to the nearest corner. This makes finding the optimal solution more difficult and requires a different approach.

- Represent the numbers in the set by a vector x, whereby x_i is the *i*th number in the set

  The problem becomes finding the solution to a linear program which also satisfies the condition xi ∈ Z for all i

- Create a boolean vector *s*, where s_i is 1 if number x_i is in the subset and 0 otherwise.
  - Convert the problem into a linear programming problem by replacing the 'is in' constraint with an inequality: si - xi >= 0
  - This will allow si to be 1 when xi is in the set and 0 otherwise.
  - Solve the linear programming problem using a solver.
  - If the optimum value of the objective function is not an integer, we can simply round it to the nearest integer.
  - If the optimum value is an integer but not all variables are 0 or 1, we can add additional constraints to force the variables to be 0 or 1. This will give us an integer solution to the problem.
  - If the problem is still not solved, we can try again with a modified objective function which penalises non-integer solutions. This is called the branch and cut method.

- Represent the target value by the constant *t*.

  We can then create a set of linear constraints: ○ For every xi, we can have a constraint si = 1 if xi <= t, or si = 0 if xi > t ○ We can also add the constraint that the sum of s must equal k, since we want to select exactly k numbers. ● We can then use a linear programming solver to find the solution to this problem. However, the solution may not necessarily have integer values for the variables si. In this case, we can use techniques such as rounding or adding additional constraints to try to force the solution to be integer. This is known as integer programming.

Minimise

$$t - \sum_{i=1}^{n} s_i x_i$$

Subject to: $\underline{\boldsymbol{s}} \in \{0,1\}$, objective function nonnegative

If the minimum value of the objective function is 0, subset sum returns *true*

Integer programming

Integer programming is NP-Hard (in fact it is NP-Equivalent), whereas linear programming problems are solvable in polynomial time.

This means that integer programming problems are at least as hard as the hardest problems in NP, and it is unlikely that there exists a polynomial time algorithm for solving them. However, there are specialized algorithms and heuristics that can often find good solutions to integer programming problems in practice, even though they may not be guaranteed to find the optimal solution.

However, because integer programming is such a common business problem, there exist powerful solvers which can often solve substantial problems in practice. The technique is *much* more effective than brute force search even though it is exponential time.

There are several ways to solve integer programming problems, including branch and bound, cutting planes, and branch and cut.

Branch and bound is a method that involves dividing the search space into smaller subproblems and solving each one individually. This is done by fixing some of the variables to specific values and then solving the resulting problem as a linear program. The solutions from these subproblems are then compared, and the best one is chosen as the final solution.

Cutting planes involves adding additional constraints to the problem to eliminate infeasible solutions. These constraints are called cuts, and they are added iteratively until a feasible solution is found.

Branch and cut is a hybrid method that combines elements of both branch and bound and cutting planes. It involves both dividing the search space into smaller subproblems and adding additional constraints to eliminate infeasible solutions.

In general, integer programming solvers use a combination of these techniques to find the optimal solution to a problem. They also often use heuristics to guide the search and improve the efficiency of the algorithm.

- The solvers commonly use a mixture of *cutting plane* techniques, *branch and cut* methods and *heuristics* to solve the problem (will elaborate in class)
- In the worst case, this produces an exponential number of subproblems to solve

  However, in practice these solvers are able to solve many large problems in reasonable time. In fact, integer programming is one of the most widely used combinatorial optimization techniques in industry, particularly in the field of operations research. It is used to solve a wide range of problems, including resource allocation, scheduling, and design problems.

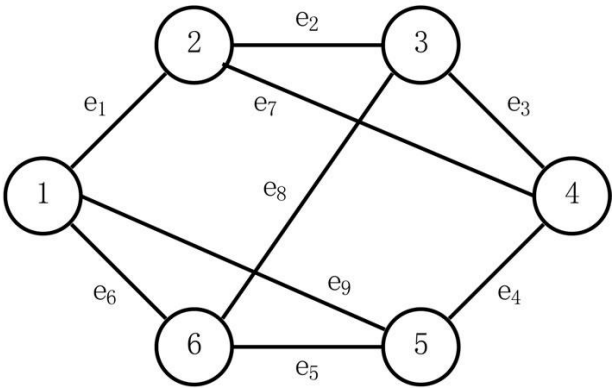
Solution techniques

So at this point we have the following items in our toolkit:

- If our new problem is solvable in polynomial time
  - Divide and conquer
  - Greedy algorithm
  - Dynamic programming
  - Linear programming
  - Etc.
- If it is NP-Complete (or NP-Equivalent)
  - Brute force search (very bad)
  - Transform it into an instance of Integer Programming and use a solver on it (might be better)
    - Because IP is NP-Equivalent, we can transform *any* NP-Equivalent problem into it!!!

Task

Use Excel to tackle this instance of the vertex cover problem:



- To recap, the vertex cover problem involves finding the minimum sized set of nodes, such that every edge is incident on at least one node of the set.
- Hint: use a binary vector to indicate whether a particular node is in the set or not.

Heuristics

Heuristics are techniques that can be used to solve problems that do not guarantee an optimal solution, but often produce good enough solutions in a reasonable amount of time. Heuristics can be used when an exact solution is not necessary or when it is not possible to solve the problem exactly within a reasonable amount of time. Some common types of heuristics include genetic algorithms, simulated annealing, and greedy algorithms.

What happens if the problem's just too big?

If a problem is too big to be solved exactly, or if we just don't have the time to wait for an exact solution, we can use heuristics to find a solution which is likely to be close to the optimal solution. Heuristics are techniques which have no guarantee of finding the optimal solution, but which work well in practice. Some common heuristics include:

- Genetic algorithms: inspired by natural evolution, these algorithms involve generating a population of candidate solutions, evaluating their quality, and then selecting the best ones to create the next generation of solutions.
- Simulated annealing: this heuristic involves starting with a randomly generated solution and then gradually refining it by making small changes and accepting or rejecting them based on a probability that decreases over time.
- Greedy algorithms: as we discussed earlier, these algorithms involve making the locally optimal choice at each step and hoping that these choices lead to a globally optimal solution.
- Too big for brute force (likely)

Too big for even the most powerful solvers

In these cases, heuristics come to the rescue. Heuristics are like algorithms, but they are not guaranteed to find the optimal solution. They are used to find good solutions quickly. Heuristics can be very effective in practice, but it is important to be aware of their limitations.

- Too big or hard to transform into Integer Programming (less likely but still common)

  Too big for a heuristic method to find the optimum solution in an acceptable amount of time

- Too big for SAT Solvers? Constraint programming? (not covered in this module but worth reading about)

  In these cases, we often turn to heuristics, which are algorithms that do not guarantee to find the optimal solution, but can often find a good solution quickly. Heuristics are often used in practice when the exact solution is not required, or when the problem is too large to solve exactly in a reasonable amount of time. Some common heuristics include simulated annealing, genetic algorithms, and local search.

What we often find ourselves doing is writing algorithms that look to find a solution efficiently but can't guarantee to return the true optimum. These processes are called *heuristics*.

Some common heuristics include:

1. Genetic algorithms: These algorithms use principles of natural selection and genetics to iteratively improve a solution.
2. Simulated annealing: This algorithm involves slowly adjusting a solution to find the global optimum, similar to the way a metal cools and solidifies.
3. Ant colony optimization: This algorithm involves simulating the behavior of ants as they search for food and use pheromone trails to guide each other to the best solution.
4. Hill climbing: This is a simple heuristic that involves starting with a random solution and iteratively improving it by making small changes until no further improvement can be found.
5. Randomized search: This involves randomly generating and evaluating potential solutions until a satisfactory one is found.

While heuristics cannot guarantee to find the optimal solution, they can often find good solutions quickly, and are often used in practice when faced with large or complex optimization problems.

An example

Let's take the vertex cover problem, since we looked at it last week. We know we almost certainly can't solve this in polytime, but how about we try something that might get us a reasonable solution most of the time.

- Select the node with the largest number of incident edges
- Remove those edges from the graph
- Repeat until there are no more edges left

One heuristic for the vertex cover problem is the following:

Start with an empty vertex cover (no vertices selected)

Select the vertex with the most number of neighbors and add it to the vertex cover

Remove all of the selected vertex's neighbors from the graph

Repeat steps 2 and 3 until all vertices have been removed or there are no more vertices with neighbors

This heuristic doesn't always produce the optimal vertex cover, but it often returns a solution that is close to the optimal solution and runs in polynomial time.

Can you find a simple instance where this works successfully?

It is not possible to find a simple instance where a heuristic solution to the vertex cover problem would always work successfully, as the vertex cover problem is NP-complete and therefore it is not possible to find an efficient solution for all instances of the problem. However, it is possible to develop heuristics that can find good approximate solutions for some instances of the problem in a reasonable amount of time.

More importantly, can you find one where it doesn't?

For example, consider the following graph:

```
  1
2   3
  4
```

The optimal vertex cover has size 2, and it can be achieved by selecting the vertices 1 and 4.

On the other hand, consider the following graph:

```
1
|
2
|
3
|
4
```

The optimal vertex cover has size 1, and it can be achieved by selecting any of the vertices 1, 2, 3, or 4. However, the heuristic of selecting the vertices with the highest degree would result in a vertex cover with size 2, as it would select vertices 1 and 2 (or 1 and 3, or 1 and 4).

Greedy heuristics

A greedy heuristic is an algorithm that makes the locally optimal choice at each step, hoping that these choices will lead to a globally optimal solution. In other words, it makes the most "appealing" choice at each step, without worrying about the consequences of future choices.

For example, a greedy algorithm for the vertex cover problem might choose the vertex with the most edges at each step, hoping that this will lead to the smallest possible vertex cover. However, this might not always lead to the optimal solution and can be used as a heuristic rather than a guaranteed solution.

The previous example is another greedy algorithm - but in this case it does not necessarily find the global optimum.

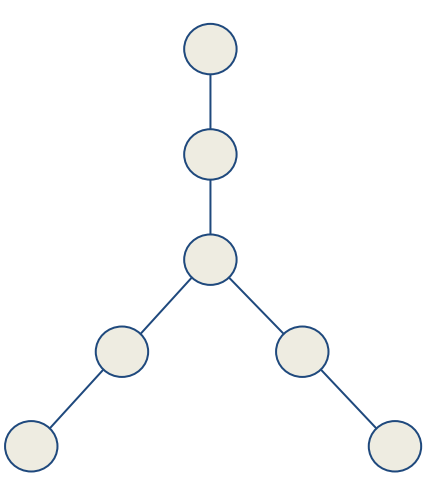
Greedy algorithms are quite common in heuristic design, as they are simple to write and can often be improved using fairly standard strategies (next week)

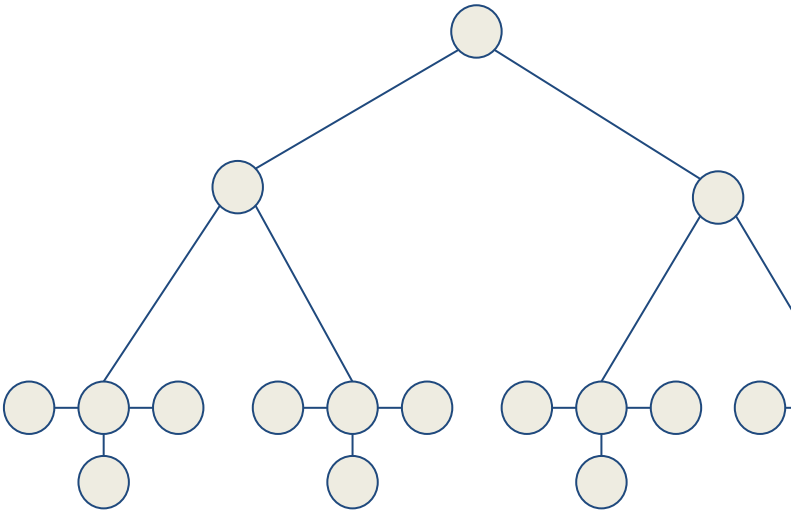
Another example for vertex cover might be:

- Start with all nodes in the vertex cover
- Remove the node with the smallest number of covered, incident edges
    - Check to see if the cover is still valid, if not, replace node and mark as 'unremovable'
- Repeat until there are no more removable nodes

Try both heuristics on these

You could use the integer program from last week to check if the results are optimal





Construction and perturbation

Construction heuristics involve building up a solution incrementally, one piece at a time. This can be done either in a greedy manner (selecting the best option at each step) or by following a specific ordering of the choices. Perturbation heuristics involve starting with an initial solution and then making small changes to it to try and improve upon it. This can be done either randomly or by following a specific set of rules. Both of these techniques can be effective for finding approximate solutions to optimization problems, but they cannot guarantee to find the true optimal solution.

There are essentially two kinds of heuristic, both of which fall under the *local search* category.

- Construction search - which starts with an empty solution and adds elements until a feasible solution is found
- Perturbation search - which starts with a feasible solution and makes small changes to it until no more improvements can be found
- The first is a construction heuristic, which starts from an empty solution and adds elements to it until a complete solution is found. A common example of this is the nearest neighbour algorithm for the travelling salesman problem, where at each step the next closest city is added to the tour.
- The second is a perturbation heuristic, which starts with a complete solution and makes local changes to it in an attempt to improve it. An example of this is the 2-opt algorithm for the travelling salesman problem, which repeatedly swaps pairs of edges in the tour to try to find a shorter route.

The first and third examples for vertex cover were construction search while the second (the drop algorithm) is a (very simple) form of perturbation search.

Construction heuristics work by building up a solution incrementally, usually by adding one element at a time. They often work by selecting the most promising element to add next according to some criterion. Perturbation heuristics, on the other hand, work by starting with an existing solution and then making small changes to it in the hope of finding a better solution. This process is repeated until no further improvements can be found.

Generic algorithms

There are also generic algorithms that can be applied to a wide range of problems, such as genetic algorithms and simulated annealing. These algorithms work by starting with a random initial solution and then iteratively making small changes to it in order to try to improve upon it. The goal is to try to find a good solution in a reasonable amount of time, even if it is not guaranteed to be the optimal solution. These algorithms are often used in practice to solve hard optimization problems where no other efficient solution is known.

Construction search:

- Start with an empty solution
    - Select an element to add, according to some criterion
    - Is the solution feasible yet?
        - If so, stop
        - Else repeat

Construction search involves starting with an empty solution and adding elements to it iteratively until we have a valid solution. This can be done in a greedy manner (choosing the best option at each step) or using a more complex procedure. An example of a construction heuristic for the vertex

cover problem is the greedy algorithm which always adds the vertex with the most incident edges to the cover. This is not guaranteed to find the optimal solution, but often performs well in practice.

Perturbation search (iterative improvement):

- Start with a feasible solution (perhaps found by a construction algorithm)
    - Make a small change to the current solution
    - Is it better?
        - If so, keep the new solution
        - Else change back
    - Repeat until no more improvements can be found

In perturbation search, we start with an initial solution and then repeatedly make small changes to it in an attempt to improve it. The changes made to the solution are called "perturbations." This process is repeated until no further improvements can be found. Perturbation search is often used in combination with local search, where the initial solution is obtained through a construction heuristic and then improved through iterative perturbations.

Construction algorithms search the space of partial solutions, looking for a feasible one, while iterative improvement algorithms search the space of feasible solutions, looking for an improvement in objective function value.

Practice

As with many things in life, practice is essential when learning to do something new. Try to produce a heuristic for each of the following problems:

- Set cover
- Travelling salesman
- Subset sum
- 3-SAT

Set Cover:

- A greedy heuristic for the set cover problem could be to repeatedly choose the set that covers the most uncovered elements and add it to the solution, until all elements are covered. This heuristic is not guaranteed to find the optimal solution, but it often performs well in practice.

Travelling Salesman:

- A common heuristic for the travelling salesman problem is the nearest neighbor algorithm, which involves starting at a random city and choosing the nearest unvisited city as the next destination until all cities have been visited. The solution is then completed by returning to

the starting city. This heuristic is not guaranteed to find the optimal solution, but it often performs well in practice.

Subset Sum:

- A simple heuristic for the subset sum problem could be to sort the elements in the set in non-decreasing order and then iteratively add the largest remaining element to the solution if it does not exceed the target sum. This heuristic is not guaranteed to find the optimal solution, but it often performs well in practice.

3-SAT:

- A heuristic for 3-SAT could be to repeatedly choose a clause with the fewest number of unassigned variables and assign values to those variables in a way that satisfies the clause. If no such assignment is possible, the heuristic could choose a random unassigned variable and flip its value. This heuristic is not guaranteed to find the optimal solution, but it often performs well in practice.

When you're done, read up on known perturbation heuristics such as 2-opt and the Lin-Kernighan algorithm for the TSP.

For the travelling salesman problem, one heuristic is called 2-opt. This heuristic works by iteratively identifying and removing any edges that are part of a sub-tour (a tour within a larger tour). These sub-tours are then replaced by a new set of edges that connect the sub-tour's endpoints directly to each other, bypassing the other vertices in the sub-tour. This process is repeated until no sub-tours can be found.

Another heuristic for the TSP is the Lin-Kernighan algorithm. This algorithm works by iteratively replacing a sequence of edges in the current tour with a shorter alternative route. This process is repeated until no further improvements can be found.

Escaping Local Optima

One of the challenges with heuristics is that they can get stuck in a local optimum, meaning that they find a solution that is good within their search space, but it may not be the global optimum (the best possible solution).

One way to try to escape local optima is to use a technique called simulated annealing, which is inspired by the way that metals are cooled and solidified. The idea is that we allow the heuristic to "explore" and potentially move to worse solutions, but with a decreasing probability as the number of iterations increases. This can help the heuristic to escape local optima and potentially find the global optimum.

Local search

Is a type of heuristic search that focuses on finding a solution that is locally optimal, rather than globally optimal. This means that the algorithm looks for the best solution within a certain range or neighborhood, rather than searching the entire space for the absolute best solution. The advantage of local search is that it can often find a good solution quickly, but the disadvantage is that it can get

stuck in a local optimum, meaning it finds a solution that is good within its neighborhood but not necessarily the best solution overall.

One way to try to escape local optima in local search is to use techniques such as simulated annealing or genetic algorithms, which allow the algorithm to make moves that may not be locally optimal in order to explore different areas of the search space and potentially find a better global solution.

In general, local search algorithms differ from exhaustive (or brute force) search in that they try to find a solution by searching only a small number of candidate solutions in order to find an answer.

This is done by starting at a solution and then making small, local changes to it in order to try to find a better solution. Local search algorithms often work by iteratively improving a solution, moving to a neighboring solution that is closer to the optimal solution. They can be seen as a form of heuristic, as they do not guarantee to find the global optimal solution, but they can often find good solutions quickly.

There are two main types of local search algorithms: construction algorithms and iterative improvement algorithms. Construction algorithms search the space of partial solutions, looking for a feasible one, while iterative improvement algorithms search the space of feasible solutions, looking for an improvement in the objective function.

Local search algorithms can be very effective for finding good solutions to hard optimization problems, but they can also get stuck in local optima, where further improvements are not possible. To escape local optima, local search algorithms can use various techniques such as randomization, perturbation, and memory.

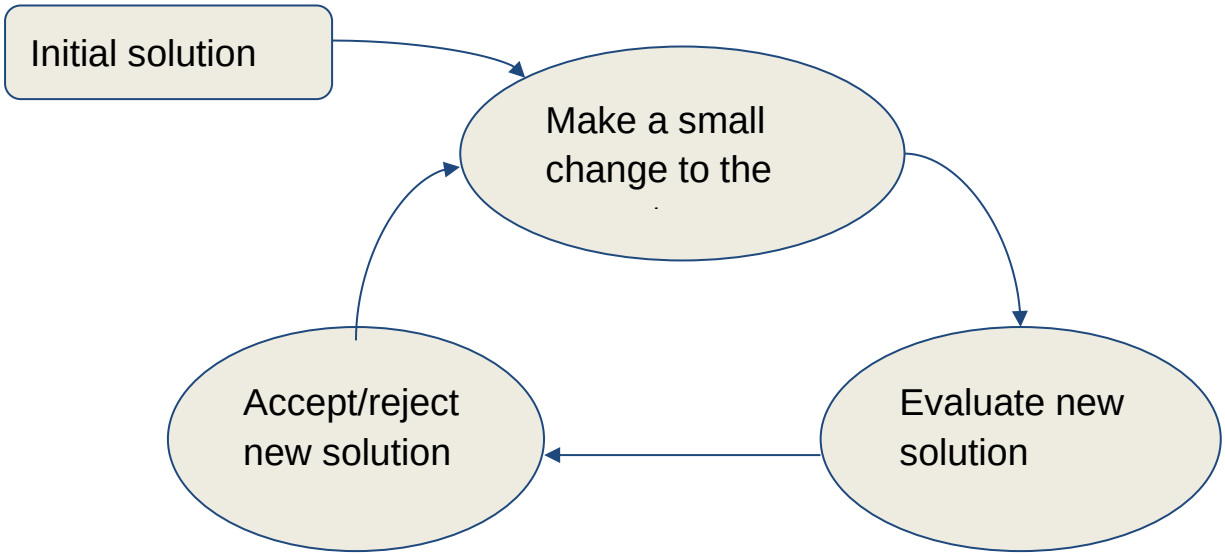
This means that, in the case of NP-Hard problems, a local search may not always guarantee to get the answer correct, but is designed to be as effective as possible most of the time.

There are a few ways that local search algorithms can escape from local optima, or suboptimal solutions:

1. Random restarts: This involves starting the search from a randomly chosen initial solution and repeating the process multiple times. This can help the algorithm escape from local optima by giving it a chance to explore different parts of the search space.
2. Perturbation: This involves intentionally introducing small changes or "perturbations" to the current solution in order to escape from local optima. This can involve randomly altering the values of some variables or swapping the values of two variables.
3. Simulated annealing: This is a probabilistic technique that can be used to escape from local optima. It is inspired by the annealing process used in metallurgy, where a material is heated and then cooled slowly in order to reduce defects and increase its structural purity. In the context of local search, simulated annealing involves allowing the algorithm to make "bad" moves with a certain probability, which can help it escape from local optima and potentially find a global optimum.
4. Genetic algorithms: This is a heuristic optimization method inspired by the process of natural evolution. It involves using principles of genetics and natural selection to evolve a population of solutions over time, with the goal of finding a global optimum. Genetic algorithms can

help escape from local optima by maintaining a diverse population of solutions and using crossover and mutation operators to explore different parts of the search space.

A generic local search process

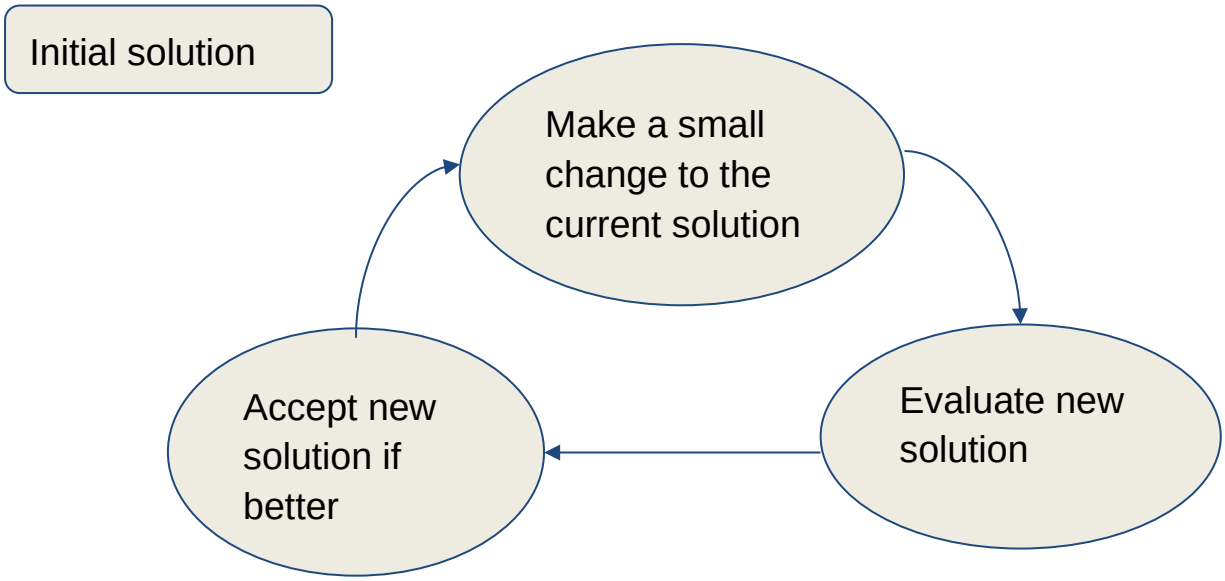


Iterative improvement

Iterative improvement algorithms are local search algorithms that start with a solution and try to find a better solution by making small changes to the current one. They continue this process until they reach a local optimum, at which point they stop. A local optimum is a solution that is better than or equal to any of its neighbours (i.e. solutions that can be reached by making a single change to the current solution).

This is the simplest form of local search technique and is commonly found as a subroutine in other more complex heuristics.

It works by starting with a feasible solution, and then repeatedly making small changes to the solution in an attempt to improve it. The process is repeated until no further improvements can be made, at which point the algorithm returns the current solution as the answer.



Example: 2-opt heuristic for the TSP

The 2-opt heuristic for the traveling salesman problem involves iteratively improving the current solution by swapping two edges that form a sub-tour with two other edges that connect the same vertices. This process is repeated until no further improvement can be made. The algorithm is called 2-opt because it only considers pairs of edges (opt is short for "optimization"). The following is a pseudocode for the 2-opt heuristic:

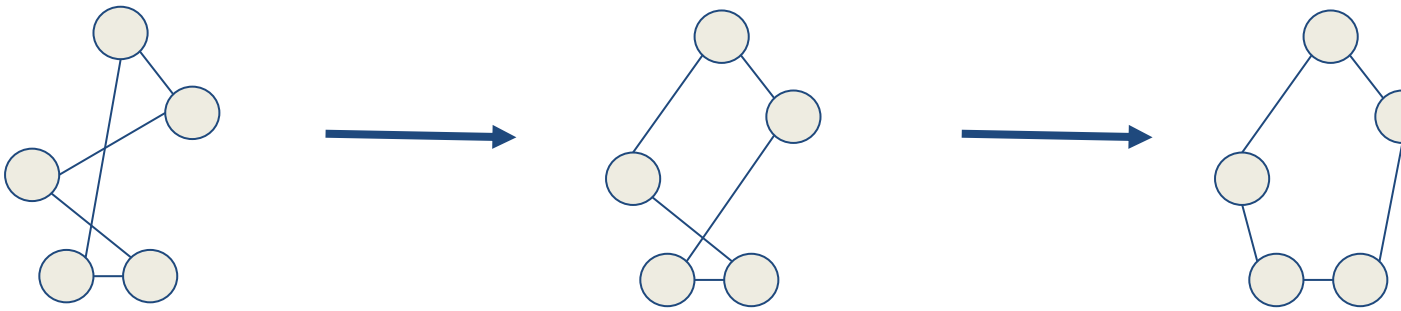
```
function 2-opt(tour):
  best_tour = tour
  improved = True
  while improved:
    improved = False
    for i in range(len(tour)):
      for j in range(i+2, len(tour)):
        new_tour = two_opt_swap(tour, i, j)
        if tour_length(new_tour) < tour_length(best_tour):
          best_tour = new_tour
          improved = True
```

return best_tour

The function `two_opt_swap(tour, i, j)` returns a new tour obtained by swapping the edges `(i, i+1)` and `(j, j+1)` in the input tour `tour`. The function `tour_length(tour)` returns the total length of the tour.

The 2-opt heuristic is a simple and effective local search algorithm for the traveling salesman problem, but it can get stuck in local optima. The Lin-Kernighan algorithm is a more advanced local search algorithm that uses the 2-opt heuristic as a subroutine and can escape local optima by making multiple 2-opt moves at a time.

In the travelling salesman problem, a common approach is to take an initial solution (perhaps created by a greedy algorithm) and repeatedly swap two edges over until no more improvement can be made. This is called a 2-opt search.



Local and Global optimality

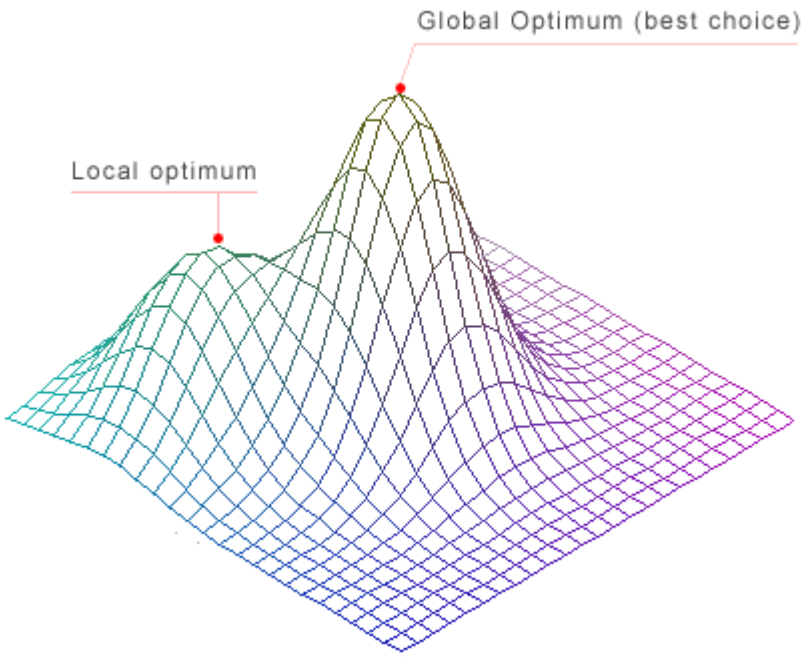
Local optimality is when a solution is the best among its neighboring solutions, while global optimality is when a solution is the best among all possible solutions. Local search algorithms tend to find local optimal solutions, while exhaustive search algorithms can find global optimal solutions. However, due to their exponential running time, exhaustive search algorithms are not practical for large problems.

When we find no further improvement using an iterative improvement algorithm, we have reached the *local optimum*. A solution which cannot be improved any further by making the small change the algorithm uses. However, the local optimum is not always equal to the global optimum.

However, this local optimum may not be the global optimum, which is the best solution possible out of all the solutions we could have found.

It is possible to get stuck at a local optimum, meaning we cannot find a better solution even though one exists. This is known as being trapped in a local optima.

To try and escape local optima and find the global optimum, we can use more complex heuristics such as genetic algorithms or simulated annealing. These techniques involve making larger changes to the current solution in order to explore more of the search space and potentially find a better solution.



Metaphors galore

There are many different metaphors that have been used to describe the concept of a local optimum. One common one is being stuck in a local "valley" where the surrounding area is lower than the current position, so there is no way to move upward. Another is being on the top of a hill and not being able to see any higher hills in the distance, so the current position is assumed to be the highest point. Regardless of the metaphor used, the idea is the same: a local optimum is a solution that cannot be improved upon by making small, local changes to it. This can be contrasted with a global optimum, which is the best solution possible out of all the possible solutions.

At this point we start to think about the shape of our problems. Linear programming problems have a favourable *search landscape*, because all local optima are global optima.

This means that if we find a solution that cannot be improved by making small changes, we know that it is the best possible solution.

Other problem types do not have such nice landscapes. For example, imagine a problem where the objective function looks like a valley with many hills. In this case, our algorithm may get stuck on a hill, even though there is a deeper valley nearby.

There are many ways to try and escape these local optima, and these methods are collectively known as metaheuristics. Some examples of metaheuristics include simulated annealing, tabu search, and genetic algorithms. These methods try to find better solutions by making larger changes to the current solution, or by trying out many different solutions simultaneously.

General optimisation problems aren’t like this

They can have multiple local optima and it's possible for the global optimum to be different from all the local optima. In these cases, iterative improvement algorithms might get stuck at a local optimum, unable to find the global one. This is called getting trapped in a local minimum.

- Imagine you want to climb to the top of a perfect cone. All you have to do is keep heading upwards and you're guaranteed to get there.
- Now imagine you want to climb to the top of a mountain. There are many different paths you can take, and you might find yourself at a local peak where you cannot climb any higher. However, if you keep exploring, you may find a path that leads you to the global peak, or the highest point on the mountain.

This metaphor illustrates the difference between local and global optima. In an optimisation problem, a local optimum is a solution that cannot be improved upon by making small changes to the solution, while a global optimum is the best possible solution to the problem. In a problem with a favourable search landscape, all local optima are also global optima, but in a problem with a more complex search landscape, there may be multiple local optima, and the global optimum may not be a local optimum.

- Does this work with a real hill (iterative improvement algorithms are often called 'hill climbers')?

No, a real hill usually has many local peaks or optima, and a hill climber may get stuck at a local peak that is not the global peak. To reach the global peak, the hill climber may need to take a step down and then climb up again, which is not possible with an iterative improvement algorithm.

Escaping local optima

There are many ways to try and escape local optima when using iterative improvement algorithms. One common method is called random restarts, where the algorithm is run multiple times with different random starting points and the best solution found is returned. Another method is called simulated annealing, where the algorithm is allowed to accept worse solutions with a certain probability in order to try and escape local optima. There are also methods that use more complex search patterns, such as genetic algorithms, which mimic the process of natural evolution to try and find good solutions.

So having established that our hill climbing (or steepest descent, if we're minimising) algorithm may leave us stuck on top of a traffic cone just outside Bangor, when trying to find the top of Snowdon, we need to think about how we escape local optima.

There are several ways to escape local optima in a search algorithm:

1. Restarting the search from a different starting point can sometimes help, as it may allow the algorithm to find a different path to the global optimum.
2. Using a more complex search algorithm, such as simulated annealing or evolutionary algorithms, which allow for the possibility of "jumping" out of a local optimum and exploring other areas of the search space.
3. Introducing randomness or noise into the search process, which can sometimes help the algorithm escape local optima and explore other areas of the search space.

4. Using a multi-start or iterated local search approach, which involves running the search algorithm multiple times from different starting points and then combining the results to find the global optimum.

Good local search algorithms need to be able to accept non-improving solutions (take a step back) to have a real chance of finding a true optimum.

One way to do this is to introduce randomness, or noise, into the search process. This can be as simple as adding a probability of accepting a non-improving solution, or by randomly perturbing the current solution and accepting it with a certain probability. This is called simulated annealing, as it is inspired by the annealing process used in metallurgy to purify and strengthen metals.

Another way to escape local optima is by using a multi-start method, where the algorithm is run multiple times from different randomly generated starting points and the best solution found is returned. This can also be combined with simulated annealing to further improve the chances of finding the global optimum.

Finally, evolutionary algorithms such as genetic algorithms use the concept of evolution and natural selection to escape local optima. These algorithms work by generating a population of candidate solutions and iteratively applying genetic operators such as crossover and mutation to generate new solutions. The best solutions are then selected to form the next generation, and the process is repeated until a satisfactory solution is found.

A good example of an algorithm that does this is *Simulated Annealing*

Simulated Annealing is a probabilistic technique used to find the global optimum of a function. It is an iterative method that starts with a randomly generated solution and iteratively makes small changes to it, in order to try to improve the solution. The algorithm accepts non-improving solutions with a certain probability, in order to avoid getting stuck in local optima. This probability is reduced over time, according to a schedule, hence the name "Simulated Annealing", which is inspired by the annealing process used in metallurgy to purify metals.

Here is an example of Simulated Annealing in Python:

```
import random
import math

def simulated_annealing(initial_solution, cost_function, neighbors_function, temperature_function):
  current_solution = initial_solution
  current_cost = cost_function(current_solution)
  for t in temperature_function:
    next_solution = random.choice(neighbors_function(current_solution))
    next_cost = cost_function(next_solution)
    delta_cost = next_cost - current_cost
    if delta_cost > 0:
      current_solution = next_solution
      current_cost = next_cost
    else:
      probability = math.exp(delta_cost / t)
      if random.random() < probability:
        current_solution = next_solution
```

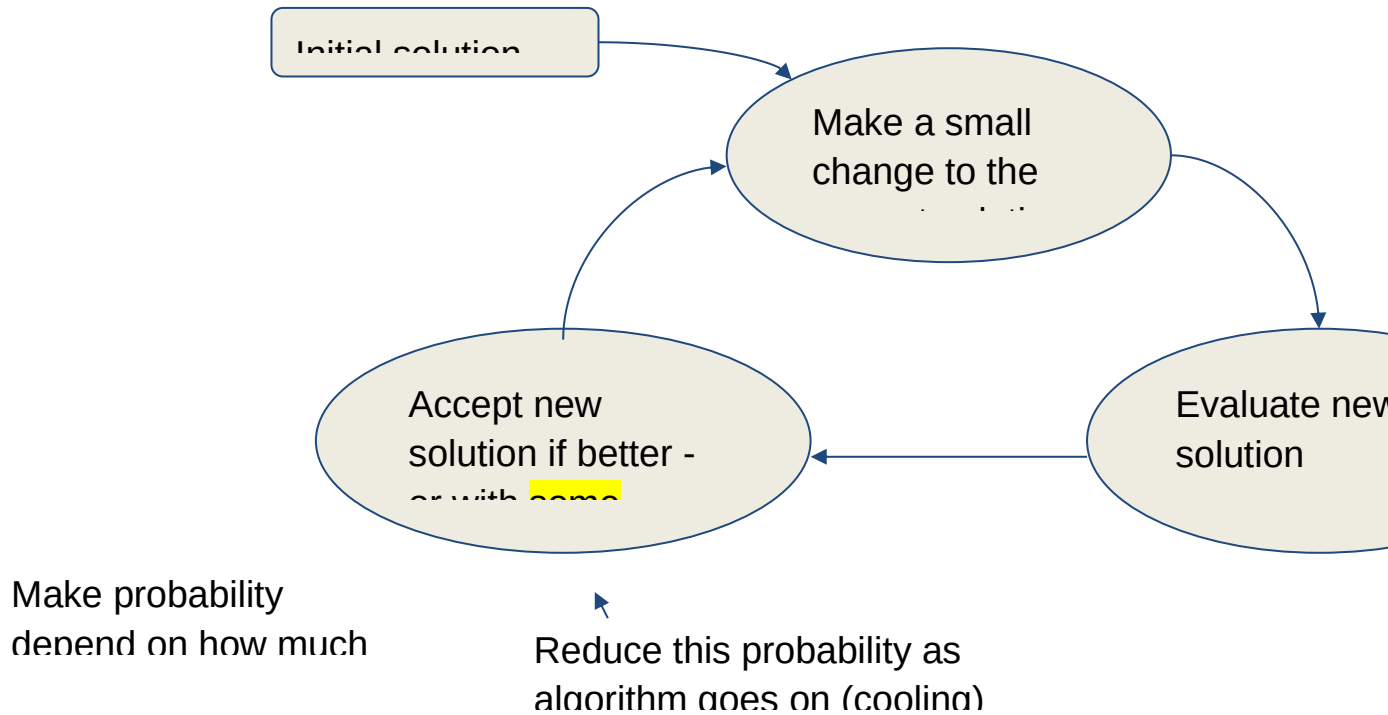
```
        current_cost = next_cost
    return current_solution
```

This algorithm takes four arguments:

- `initial_solution`: the initial solution generated by the algorithm.
- `cost_function`: a function that takes a solution and returns its cost.
- `neighbors_function`: a function that takes a solution and returns a list of its neighbors (i.e., solutions that can be obtained by making small changes to it).
- `temperature_function`: a function that generates a sequence of temperatures that are used to control the probability of accepting non-improving solutions.

The algorithm iteratively generates a new solution from the current solution's neighbors, and accepts it as the new current solution if its cost is lower, or with a certain probability if its cost is higher. The probability of accepting a higher-cost solution is controlled by the current temperature, which is reduced over time according to the temperature function. This way, the algorithm initially has a higher chance of accepting non-improving solutions, in order to explore the search space more widely, and then reduces this probability over time, in order to converge to the global optimum.

Simulated Annealing (SA)



The SA algorithm

Simulated Annealing (SA) is a local search algorithm that is used to find a good approximate solution to a problem that is difficult to solve exactly. It is based on the idea of annealing in metallurgy, where a material is heated and then cooled slowly in order to reduce defects and increase its structural purity.

The algorithm works by simulating the process of annealing in a material by starting at a high temperature and gradually decreasing it over time. At high temperatures, the algorithm is more likely to accept non-improving solutions, which allows it to explore a larger portion of the search space and potentially escape local optima. As the temperature decreases, the algorithm becomes more selective and is less likely to accept non-improving solutions, which helps it to converge on a better solution.

Here is a simple example of how the SA algorithm might be implemented in Python:

```
import random

def simulated_annealing(problem, max_iterations, temperature_function):
 current_solution = problem.get_initial_solution()
 best_solution = current_solution

 for i in range(max_iterations):
  temperature = temperature_function(i)

  next_solution = problem.get_random_neighbor(current_solution)
  delta_e = problem.get_value(next_solution) - problem.get_value(current_solution)

  if delta_e > 0:
   current_solution = next_solution
   if problem.get_value(current_solution) > problem.get_value(best_solution):
    best_solution = current_solution
  else:
   probability = math.exp(delta_e / temperature)
   if random.random() < probability:
    current_solution = next_solution

 return best_solution
```

This implementation of the SA algorithm takes as input a `problem` object that represents the problem to be solved, the maximum number of iterations to run the algorithm for, and a `temperature_function` that returns the temperature at each iteration. The `problem` object should have methods for getting an initial solution, generating a random neighbor of a given solution, and computing the value of a solution.

The algorithm starts by getting an initial solution and setting the best solution found so far to this initial solution. It then iterates for a given number of times, getting the temperature at each iteration using the temperature function and generating a random neighbor of the current solution. If the value of the neighbor is greater than the value of the current solution, the current solution is updated to be the neighbor and the best solution is updated if the current solution is better than the best solution found so far. If the value of the neighbor is not greater than the value of the current solution, the current solution is still updated to be the neighbor with a probability equal to `exp(delta_e / temperature)`, where `delta_e` is the difference in value between the neighbor and the current solution. This probability allows the algorithm to occasionally accept non-improving solutions, especially at high temperatures. After the maximum number of iterations has been reached, the algorithm returns the best solution found.

Simulated annealing works just like iterative improvement, but with a modification to the acceptance criterion. It uses the *metropolis criterion*, derived from the cooling (annealing) of molten crystals. A new solution is accepted in two circumstances

- If the new solution is better (like iterative improvement)
- If it is worse and $r < \exp(\delta/T)$
    - r is a random number between 0 and 1
    - δ is the difference in evaluation between the old and new solutions (should always be negative: old-new for minimisation, new-old for maximisation).
    - T is a variable called the *temperature*, which reduces as the algorithm continues - this makes the process more likely to accept non-improving solutions at the beginning and less likely towards the end. Usually by multiplying by a value in the range 0.9 - 0.99, periodically.

Tabu Search

Tabu search is a local search algorithm that uses a "memory" of past moves to guide the search process. It is designed to escape local optima by allowing the algorithm to move to a non-optimal solution if it has not been visited in the recent past. This helps to prevent the algorithm from getting stuck in a local optimum, and can also help to prevent cycling. The algorithm maintains a list of "tabu" moves, which are moves that are not allowed to be made for a certain number of iterations. This list is updated at each iteration, and the algorithm tries to find a move that is not tabu and that improves the solution. The use of a tabu list helps to prevent the algorithm from getting stuck in a cycle and allows it to explore a wider range of the search space.

An alternative approach to escape local optima is given by the Tabu Search framework.

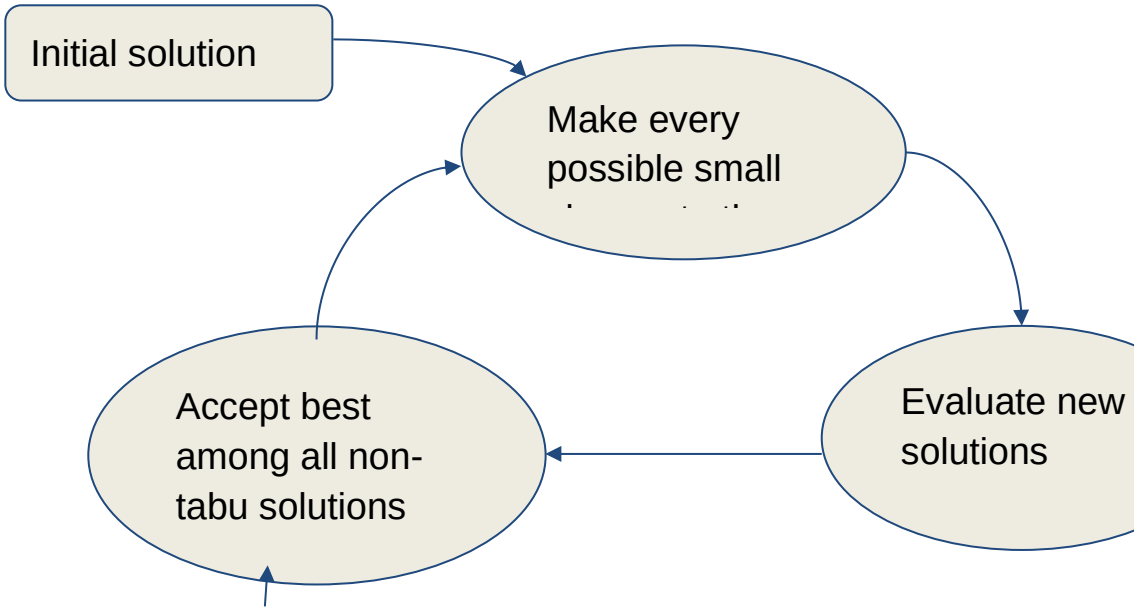
- Rather than picking any improving solution at each iteration, always pick the best (even if it is not an improvement).
- Remember which choices you made recently and do not unmake them (stops you climbing straight back up)

Iteratively try to improve the solution by making a move (swap two items, remove one, add one etc.) and check if it is an improvement ● If it is, accept it ● If it isn't, accept it with some probability (controlled by a temperature parameter, which decreases over time). This probability should be high when the temperature is high and low when the temperature is low. ● Update the temperature according to a schedule (e.g. linear decrease, exponential decrease) ● Repeat until the temperature is low enough or a satisfactory solution is found

This approach uses *adaptive memory* to keep a list of recently altered components. These components are marked as *tabu* (or *taboo* - as in forbidden) for a set number of iterations (the *tabu tenure*) before they can be changed again.

The hope is that by restricting the search to new solutions, the algorithm will be able to escape local optima and find better solutions. Tabu search has been very successful in practice and is used in a wide range of applications.

Tabu Search (TS)



May use *aspiration criteria* to accept a non tabu solution if it is very good - such as a new optimum.

Metaheuristics

Metaheuristics are high-level heuristics that aim to guide other heuristics towards better solutions. They are often used to solve large-scale, complex optimization problems where the search space is too vast to be searched exhaustively. Metaheuristics typically work by iteratively improving a candidate solution, trying to escape local optima and find a global optimum. Some examples of metaheuristics include simulated annealing, tabu search, and evolutionary algorithms.

Metaheuristics

- A metaheuristic is a master strategy used to guide a simpler heuristic search for a solution
    - For example, a simple 2-opt heuristic can be greatly improved by adapting it using simulated annealing or tabu search.
    - Further metaheuristics used to escape local optima are
        - Variable neighbourhood search
        - Guided local search
        - Iterated local search

Evolutionary algorithms (EA) are another type of metaheuristic. These algorithms are inspired by the process of natural evolution and use mechanisms such as reproduction, mutation, and selection to search for solutions to problems. EAs are commonly used to solve optimization problems and are particularly useful for problems with a large search space and no known algorithm for finding an optimal solution. Examples of evolutionary algorithms include genetic algorithms, differential evolution, and particle swarm optimization.

Construction metaheuristics

Construction metaheuristics are methods that iteratively build up a solution to a problem, usually starting from a partially constructed or empty solution and adding new elements until a complete solution is obtained. Some examples of construction metaheuristics include:

- Greedy algorithms: These algorithms make the locally optimal choice at each step, hoping that these choices will lead to a globally optimal solution.
- Genetic algorithms: These algorithms use principles of natural evolution, such as selection, crossover, and mutation, to generate new solutions and improve upon existing ones.
- Ant colony optimization: This algorithm simulates the behavior of ants searching for food, using pheromone trails left by other ants to guide their search.

Construction metaheuristics are often used to solve problems in which the solution can be built up incrementally, such as scheduling or routing problems. They are typically less effective at solving optimization problems that require a significant amount of modification to an existing solution in order to improve it.

We can use metaheuristics to adapt greedy, construction algorithms as well as perturbation searches.

One example of a construction metaheuristic is the greedy randomized adaptive search procedure (GRASP). This algorithm starts by constructing a solution using a greedy approach, but then introduces randomness to escape local optima. At each step of the construction process, a set of candidates is created and a random candidate is chosen to be included in the solution. The algorithm then continues the construction process until a complete solution is formed. GRASP can be used to solve a variety of optimization problems, including the knapsack problem and the traveling salesman problem.

Let's take an adaptation of the nearest neighbour algorithm for the TSP.

The nearest neighbour algorithm is:

- Start at a given node
- Find the node nearest to it and add to the tour
- Then keep picking the nearest unused node and add until there are no more nodes left.

This can produce some pretty bad results - but we can improve it using a metaheuristic...

Here is an example of how we can use the metaheuristic of simulated annealing to improve the nearest neighbor algorithm for the TSP:

```
import random

def calc_distance(tour):
  # Calculate the total distance of a given tour
  distance = 0
  for i in range(len(tour) - 1):
    distance += distance_matrix[tour[i]][tour[i + 1]]
  return distance

def simulated_annealing(tour, temperature, cooling_rate):
  # Improve the given tour using simulated annealing
  while temperature > 0.1:
    # Pick two random nodes to swap
    i = random.randint(0, len(tour) - 1)
    j = random.randint(0, len(tour) - 1)
    if i == j:
      continue

    # Calculate the current and new distances
    current_distance = calc_distance(tour)
    new_distance = calc_distance(tour[:i] + tour[i:j + 1][::-1] + tour[j + 1:])
    delta = new_distance - current_distance

    # If the new tour is better, accept it
    if delta < 0:
      tour = tour[:i] + tour[i:j + 1][::-1] + tour[j + 1:]
    else:
      # If the new tour is worse, accept it with a probability
      # proportional to the current temperature
      if random.random() < 2.71828 ** (-delta / temperature):
        tour = tour[:i] + tour[i:j + 1][::-1] + tour[j + 1:]
```

```
    # Decrease the temperature
    temperature *= 1 - cooling_rate
  return tour

# Initialize variables
nodes = [0, 1, 2, 3, 4]
distance_matrix = [[0, 10, 15, 20, 25],
                   [10, 0, 35, 25, 30],
                   [15, 35, 0, 30, 15],
                   [20, 25, 30, 0, 40],
                   [25, 30, 15, 40, 0]]
tour = [0, 1, 2, 3, 4]

# Run the simulated annealing algorithm
tour = simulated_annealing(tour, 100, 0.1)

# Print the final tour and distance
print("Final tour:", tour)
print("Distance:", calc_distance(tour))
```

This code first defines a function `calc_distance` that calculates the total distance of a given tour. Then it defines the `simulated_annealing` function, which takes in a tour, a starting temperature, and a cooling rate. It repeatedly swaps two random nodes in the tour and calculates the new distance. If the new tour has a shorter distance, it is always accepted. If the new tour has a longer distance, it is accepted with a probability proportional to the current temperature. The temperature is decreased at each iteration by the cooling rate. Finally, the code initializes some variables and runs the `simulated_annealing` function to improve the initial tour.

GRASP

GRASP (Greedy Randomized Adaptive Search Procedure) is a metaheuristic for combinatorial optimization problems that combines the benefits of greediness and randomization. It works by iteratively constructing a solution by greedily selecting the locally best available option and then randomly perturbing the solution to escape local optima.

The algorithm works as follows:

1. Initialize an empty solution
2. While the solution is not complete:
3. Select the best available option using a greedy heuristic
4. Randomly perturb the solution by making a small number of changes
5. Return the final solution

The key idea behind GRASP is to use the greediness of a local search algorithm to quickly find a good solution, while using the randomization to escape local optima and improve the overall quality of the solution.

The GRASP, or *Greedy Random Adaptive Search Procedure* metaheuristic takes a greedy algorithm, randomises its choices from a *Reduced Candidate List* (RCL) and repeats.

- Repeat until no improvement found in *k* passes

- Start at a random node
- While there are unused nodes
  - Select the nearest *c* neighbours from the tour's endpoints
  - Randomly pick one and add to the tour
- (optional) do 2-opt search on resulting tour

We can perform a much more intensive search using this method, and keep getting different results each time.

Task

Try to produce a GRASP algorithm for the vertex cover problem by adapting one of the greedy algorithms.

To create a GRASP algorithm for the vertex cover problem, we can start by adapting the greedy algorithm that iteratively adds the vertex with the highest degree to the cover until all edges are covered.

Here is an example of how this could be implemented in Python:

```
def grasp_vertex_cover(G):
  # G is the input graph in the form of a list of edges

  # Initialize an empty vertex cover
  vertex_cover = set()

  # Repeat the following process a number of times (we can experiment with different values for the number of iterations)
  for i in range(10):
    # Create a list of the remaining vertices that are not yet in the cover
    remaining_vertices = set(G.keys()) - vertex_cover

    # Initialize an empty candidate set for the current iteration
    candidate_set = set()

    # While there are still remaining vertices
    while len(remaining_vertices) > 0:
      # Find the vertex with the highest degree
      highest_degree_vertex = max(remaining_vertices, key=lambda v: len(G[v]))

      # Add the highest degree vertex to the candidate set and remove it from the list of remaining vertices
      candidate_set.add(highest_degree_vertex)
      remaining_vertices.remove(highest_degree_vertex)

      # Remove all the edges incident to the highest degree vertex from the graph
      for neighbor in G[highest_degree_vertex]:
        del G[neighbor][highest_degree_vertex]

    # Choose the best solution from the candidate set using a randomization procedure (we can use a random selection or a greedy selection based on the vertex degrees)
    vertex_cover.update(random.sample(candidate_set, k=1))
```

```
    # Return the final vertex cover
    return vertex_cover
```

This GRASP algorithm for the vertex cover problem works by repeating the greedy algorithm a number of times, each time constructing a candidate set of vertices using the greedy selection strategy. The final vertex cover is then chosen from the union of all the candidate sets using a randomization procedure. This allows the algorithm to escape local optima and potentially find a better overall solution.

Ant Colony Optimisation (ACO)

Ant Colony Optimisation (ACO) is a heuristic optimization algorithm that simulates the behavior of ants searching for food. It is often used to find solutions to combinatorial optimization problems, such as the travelling salesman problem or the knapsack problem.

ACO works by maintaining a population of "virtual ants" that iteratively build solutions to the optimization problem by constructing a path through a graph. At each step, an ant chooses the next node in its path based on a combination of the pheromone levels on the edges and the distance between the nodes. As the ants traverse the graph, they leave behind a trail of pheromones that influence the choices of the other ants. Over time, the pheromone levels on the edges of the graph will converge towards the optimal solution, as the ants are more likely to follow paths with higher pheromone levels.

Here is a simple example of ACO implemented in Python for solving the travelling salesman problem:

```
import random

# Number of cities
n = 20

# Distance matrix
dist = [[0] * n for _ in range(n)]

# Randomly generate distance matrix
for i in range(n):
    for j in range(n):
        if i == j:
            dist[i][j] = 0
        elif dist[i][j] == 0:
            dist[i][j] = dist[j][i] = random.randint(1, 10)

# Number of ants
m = 10

# Number of iterations
t = 100

# pheromone matrix
pheromone = [[1 / (n * n)] * n for _ in range(n)]

# Heuristic information matrix
heuristic = [[1 / dist[i][j] for j in range(n)] for i in range(n)]

# Evaporation rate
```

```
q = 0.5

# Intensity of pheromone deposit
d = 0.1

# Find the shortest path using ACO
best_path = float("inf")
best_tour = []
for _ in range(t):
    tours = []
    for k in range(m):
        tour = [random.randint(0, n - 1)]
        while len(tour) < n:
            last = tour[-1]
            choices = [i for i in range(n) if i not in tour]
            probs = [pheromone[last][i] ** q * heuristic[last][i] ** (1 - q) for i in choices]
            norm = sum(probs)
            probs = [p / norm for p in probs]
            tour.append(choices[weighted_choice(probs)])
        tours.append(tour)

    for tour in tours:
        length = sum([dist[tour[i]][tour[i + 1]] for i in range(-1, n - 1)])
        if length < best_path:
            best_path = length
            best_tour = tour
        for i in range(-1, n - 1):
            pheromone[tour[i]][tour[i + 1]] += d / length
            pheromone[tour[i + 1]][tour[i]] += d / length
    for i in range(n):
        for j in range(n):
            pheromone[i][j] *= 1 - q

print(best_path)
print(best_tour)
```

This ACO algorithm works by simulating the behavior of ants searching for food. In this case, each ant is searching for the shortest path between a set of cities. At each step, the ant chooses the next city to visit based on the pheromone levels on the edges and the heuristic value of the cities. Pheromones are a chemical trail left by ants that other ants can follow, and in this case, the pheromone levels represent the desirability of an edge. The heuristic value represents the distance between two cities, with shorter distances being more desirable.

The algorithm runs for a specified number of iterations, with each iteration consisting of multiple ants building tours. At the end of each iteration, the pheromone levels on the edges are updated based on the length of the tours and the amount of pheromone deposited by the ants. The pheromone update rule used in this example is:

pheromone[i][j] = (1-rho)*pheromone[i][j] + delta_pheromone[i][j]

Where rho is the pheromone evaporation coefficient (set to 0.1 in this example) and delta_pheromone[i][j] is the amount of pheromone deposited on edge (i,j) by the ants. This amount is proportional to the length of the tour and the quality of the solution, with shorter and better tours depositing more pheromone.

The probability of an ant choosing a particular edge is also influenced by the pheromone levels on that edge. An edge with high pheromone levels will be more attractive to the ants, while an edge with low pheromone levels will be less attractive. The probability of an ant choosing edge (i,j) is given by:

probability[i][j] = (pheromone[i][j]**alpha) * ((1.0/distance[i][j])**beta)

Where alpha and beta are parameters that control the relative importance of the pheromone and distance factors. In this example, alpha is set to 1 and beta is set to 2.

The ACO algorithm iterates for a specified number of iterations, with each iteration consisting of the following steps:

1. Initialise pheromone levels on all edges to a small positive value.
2. For each ant:
    1. Initialise the ant at a random node.
    2. While the ant is not at the final node:
        1. Choose the next node based on the probabilities of the available edges.
        2. Update the ant's tour length.
        3. Deposit pheromone on the chosen edge.
    1. Update the best tour length and the corresponding tour if the ant's tour is better.
1. Update the pheromone levels on all edges using the pheromone update rule.
2. Repeat from step 2 until the number of iterations is reached.

At the end of the algorithm, the best tour found by the ants is returned as the solution.

ACO is a metaheuristic based on the foraging behaviour of Argentine ants.

- It is observed that ant colonies are good at finding quick routes to food sources on a tree.
- They achieve this by a process called *stigmergy*
    - Ants explore the tree randomly but leave pheromone trails when they return from a food source
    - The pheromone trail is stronger the nearer the food source is
    - Subsequent ants are still random, but bias their choices in favour of the branches with strong pheromone trails
    - The good trails get stronger and eventually all ants converge on the best route - until the food runs out.
    - This has actually been simulated for large scale network routing and outperforms many standard approaches

ACO for the TSP

We could do a simple adaptation for the TSP as follows:

- Start with all pheromone levels equal (or zero)
- While there is no improvement in *k* iterations:
    - Send out a number of ants (in parallel ideally)
    - For each ant
        - while there are unused nodes
            - Choose an edge randomly to join the tour, based on pheromone distribution
        - Do 2-opt or other iterative improvement search (optional)
        - Update pheromone levels so that highest levels are on edges in shortest tours

Evaporate (remove a percentage of pheromones from all edges)

Population vs Trajectory based techniques

Population based techniques refer to algorithms that maintain a population of candidate solutions and use some form of selection and variation to improve the solutions over time. Examples of population based techniques include evolutionary algorithms and genetic algorithms.

Trajectory based techniques, on the other hand, refer to algorithms that follow a single candidate solution as it evolves over time. These algorithms do not maintain a population of solutions and instead only focus on a single solution at a time. Examples of trajectory based techniques include simulated annealing and tabu search.

This is our first example of a *population* based technique, which works on many candidate solutions at a time and not just one.

Other examples of population based techniques include genetic algorithms, particle swarm optimization, and ant colony optimization. These techniques are often inspired by natural processes and involve the manipulation and evolution of a group of potential solutions.

On the other hand, trajectory based techniques involve the manipulation of a single solution, which is iteratively improved upon until it reaches an optimal or satisfactory state. Examples of trajectory based techniques include local search algorithms such as simulated annealing, tabu search, and iterated local search. These techniques work by making small changes to a current solution and evaluating whether it results in an improvement. If it does, the solution is accepted and the process continues, otherwise the solution is rejected and the process returns to the previous state.

These are particularly useful as they lend themselves well to parallel computation.

Genetic algorithms (GAs) are another example of population based techniques. They work by creating a population of candidate solutions (often called chromosomes), which are then evolved over time using the principles of natural selection and genetics.

In each iteration, the fitness of each chromosome is evaluated and the best performing ones are selected to create a new population of offspring through processes such as crossover (recombination of genes) and mutation. This process is repeated until a satisfactory solution is found or a pre-determined number of iterations have been reached.

GAs are often used to solve optimization problems, but they can also be applied to problems in machine learning and data mining. They are particularly useful for problems where the search space is large and the objective function is complex and non-linear.

Here is an example of a simple genetic algorithm implemented in Python for solving the knapsack problem:

```
Initialize population
population = [Individual() for i in range(POPULATION_SIZE)]
Evaluate population
for individual in population: individual.fitness = evaluate_fitness(individual.chromosome)
Loop until termination condition is met
while not termination_condition_met(): # Select parents parents = select_parents(population)
# Generate offspring
offspring = []
for i in range(0, len(parents), 2):
    offspring += crossover(parents[i], parents[i+1])

# Mutate offspring
for i in offspring:
    mutate(i)

# Evaluate offspring
for individual in offspring:
    individual.fitness = evaluate_fitness(individual.chromosome)

# Replace population with offspring
population = offspring
Return the best individual in the final population
best_individual = sorted(population, key=lambda x: x.fitness, reverse=True)[0] return best_individual
```

Genetic Algorithms and Evolutionary Computing

Genetic algorithms are a type of optimization algorithm that are inspired by the process of natural selection and evolution in biology. They are used to find solutions to optimization problems by simulating the process of evolution in a population of candidate solutions.

The basic idea behind genetic algorithms is to start with a population of candidate solutions, known as "chromosomes," and apply a set of rules to generate a new population of chromosomes for each iteration of the algorithm. These rules are inspired by the mechanisms of natural evolution, such as reproduction, mutation, and natural selection.

In each iteration, the genetic algorithm selects the best-performing chromosomes from the current population and reproduces them to create a new population of offspring. It also introduces random mutations to the offspring to add diversity to the population and prevent it from getting stuck in a local minimum. Finally, it applies a selection process to choose the fittest chromosomes to survive and carry on to the next generation.

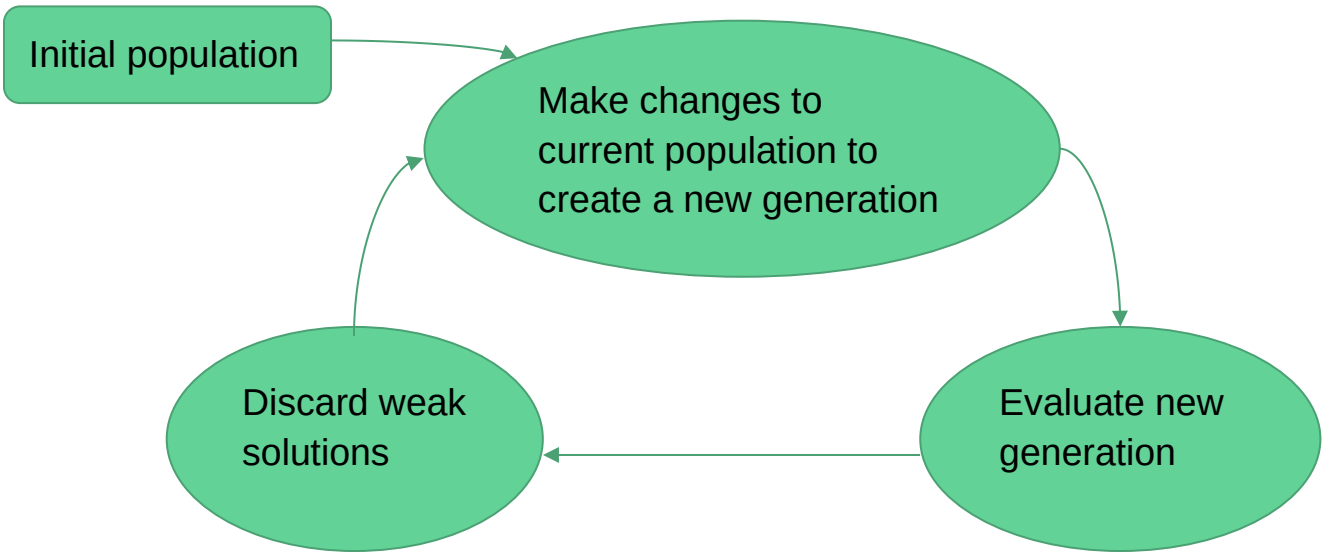
The process is repeated until the algorithm reaches a satisfactory level of performance or a predefined number of iterations has been reached. At this point, the best-performing chromosome in the final population is returned as the solution to the optimization problem.

Genetic algorithms are widely used in various fields, including machine learning, data analysis, and engineering, to find solutions to a wide range of optimization problems. They are particularly useful for problems that have a large search space and are difficult to solve using traditional optimization techniques.

Genetic algorithms are similar to local search but with two main differences

- GAs are population based so operate on many solutions at a time
- They allow for *crossover* and *recombination* of candidate solutions as well as *mutation*
    - Our algorithms up until now have allowed only mutation, applying a small alteration to a single solution
    - Crossover and recombination allow the creation of new solutions by combining characteristics of existing ones (think parents and children here)
    - Genetic algorithms traditionally operated only on solutions represented by bitstrings, a similar process using a more complex solution (eg. containing integers) is called an *evolutionary algorithm*.
    - Many GAs only involve mutation as crossover is tricky and often breaks feasibility

GAs



Ridiculous metaphor

One way to think about genetic algorithms is to imagine that you are trying to climb a very tall mountain. You are not sure exactly where the peak is, but you have a pretty good idea of the general direction. You also know that the mountain is very steep, and there are a lot of cliffs and other obstacles that will make it difficult to reach the top.

To climb the mountain, you have a group of people (called the population). Each person has a unique set of traits that will help them navigate the mountain and reach the peak. These traits might include things like strength, agility, endurance, and so on.

As the group begins to climb, some people will naturally do better than others. The stronger and more agile people will be able to climb faster and more efficiently, while the weaker and less agile people will struggle.

Over time, the group will naturally evolve. The stronger and more agile people will reproduce and pass on their traits to their offspring, while the weaker and less agile people will either die off or be left behind.

As the group continues to evolve and climb the mountain, they will eventually reach the peak. At this point, the group will have the optimal set of traits for navigating the mountain and reaching the top.

This is essentially how genetic algorithms work. The mountain represents the problem you are trying to solve, and the population represents the different possible solutions. Through a process of evolution and natural selection, the population will eventually find the best solution to the problem.

- Iterative improvement is like getting a kangaroo to climb a hill by always going upwards. It very quickly gets stuck on top of a small hump.
- Simulated annealing gets the kangaroo drunk first, so it staggers around with a bit of an upwards bias but eventually sobers up and starts acting like iterative improvement again.
- Tabu search allows the kangaroo to go down if there is no other alternative and gives it an electric shock if it tries to go back on itself.
- Genetic algorithms have loads of kangaroos, let them walk whenever they want but periodically shoot any kangaroos going the wrong way.
- Ant colony optimisation has many kangaroos which walk around randomly but start shrieking as they climb higher. Subsequent kangaroos follow the noise.

Hyperheuristics

Hyperheuristics are higher level strategies used to guide the selection of heuristics for solving problems. They can be seen as meta-metaheuristics, in that they are used to choose between different metaheuristics for different problems. The goal of hyperheuristics is to find the most effective heuristic for a given problem, rather than trying to use a single heuristic to solve all problems. Hyperheuristics can be particularly useful in situations where it is not clear which heuristic will be most effective for a particular problem.

Hyperheuristics are high-level strategies that guide the selection and configuration of lower-level heuristics. In other words, they are meta-heuristics that operate on other heuristics. Hyperheuristics are used to solve complex optimization problems where it is not possible to design a specific heuristic for the problem at hand.

There are two main types of hyperheuristics:

1. Population-based hyperheuristics: These hyperheuristics operate on a population of solutions and use genetic algorithms, evolutionary algorithms, or other population-based optimization techniques to find good solutions.
2. Trajectory-based hyperheuristics: These hyperheuristics operate on a single solution and use local search, iterative improvement, or other trajectory-based optimization techniques to find good solutions.

Some examples of hyperheuristics include:

1. Hybrid genetic algorithms: These algorithms combine genetic algorithms with other heuristics such as local search or simulated annealing to find good solutions.
2. Hybrid evolutionary algorithms: These algorithms combine evolutionary algorithms with other heuristics such as local search or simulated annealing to find good solutions.
3. Hyper-heuristic selection algorithms: These algorithms select the most appropriate heuristic for a given problem based on the characteristics of the problem and the performance of the heuristics on previous problems.

Leading thinkers in the field of hyperheuristics include Carsten Witt, Edmund Burke, and Graham Kendall. Some key works on the subject include "Hyper-Heuristics: An Emerging Direction in Modern Search Technology" by Carsten Witt and "Hyper-Heuristics: A Survey of the State of the Art" by Edmund Burke and Graham Kendall.

Hyperheuristics are metaheuristics that are used to guide the selection and/or generation of heuristics for specific problems. They are often used in cases where it is difficult to determine which heuristic will work best for a particular problem, or when the problem is too complex to be solved using a single heuristic. Some examples of hyperheuristics include:

1. Hybrid Genetic Algorithms: These algorithms combine genetic algorithms with other heuristics, such as simulated annealing or tabu search, to improve their performance.
2. Portfolio-Based Hyperheuristics: These algorithms use a portfolio of heuristics, rather than a single heuristic, to solve a problem. The heuristics in the portfolio are selected based on their performance on a set of test problems, and the hyperheuristic selects the best heuristic for a given problem based on its past performance.
3. Hyperheuristics Based on Machine Learning: These algorithms use machine learning techniques, such as decision trees or neural networks, to learn which heuristics are most effective for solving a particular problem.

Some of the leading thinkers in the field of hyperheuristics include Kenneth De Jong, Peter Ross, and Edmund Burke. Some key works in this field include "Hyperheuristics: An Emerging Direction in Modern Heuristics" by Edmund Burke and "Hyper-Heuristics: A Survey of the State of the Art" by Kenneth De Jong and Peter Ross.

Hyperheuristics are metaheuristics that are used to guide the selection and/or generation of heuristics for specific problems. They are often used in cases where it is difficult to determine which heuristic will work best for a particular problem, or when the problem is too complex to be solved using a single heuristic. Some examples of hyperheuristics include:

1. Hybrid Genetic Algorithms: These algorithms combine genetic algorithms with other heuristics, such as simulated annealing or tabu search, to improve their performance.
2. Portfolio-Based Hyperheuristics: These algorithms use a portfolio of heuristics, rather than a single heuristic, to solve a problem. The heuristics in the portfolio are selected based on their performance on a set of test problems, and the hyperheuristic selects the best heuristic for a given problem based on its past performance.
3. Hyperheuristics Based on Machine Learning: These algorithms use machine learning techniques, such as decision trees or neural networks, to learn which heuristics are most effective for solving a particular problem.

Some of the leading thinkers in the field of hyperheuristics include Kenneth De Jong, Peter Ross, and Edmund Burke. Some key works in this field include "Hyperheuristics: An Emerging Direction in Modern Heuristics" by Edmund Burke and "Hyper-Heuristics: A Survey of the State of the Art" by Kenneth De Jong and Peter Ross.

Hyperheuristics are high-level strategies that can be applied to a wide range of problems and can guide the selection and execution of lower-level heuristics. They are designed to handle complex and dynamic environments, where it is difficult to manually design effective heuristics.

Hyperheuristics can be classified into three categories: selection, generation, and hybrid. Selection hyperheuristics select an appropriate heuristic from a set of pre-defined heuristics based on the problem characteristics. Generation hyperheuristics generate a new heuristic specifically tailored to the problem at hand. Hybrid hyperheuristics combine selection and generation approaches.

There are many different hyperheuristic approaches, including:

- Learning classifier systems: these use machine learning techniques to select and generate heuristics
- Genetic algorithms: these use evolutionary computation to search for good heuristics
- Ant colony optimization: these use the behavior of ants to guide the search for heuristics
- Artificial neural networks: these use a neural network to learn a mapping from problem characteristics to effective heuristics

Hyperheuristics have been applied to a variety of problem domains, including scheduling, resource allocation, and vehicle routing. Some of the leading researchers in the field of hyperheuristics include Edmund Burke, Andries Petrus Engelbrecht, and Michel Gendreau. Key works in the field include the book "Hyperheuristics: An emerging direction in modern search technology" edited by Edmund Burke and Graham Kendall, and the journal "Journal of Heuristics" edited by Edmund Burke and Andries Petrus Engelbrecht.

"Hyper-Heuristics: An Emerging Direction in Modern Search Technology" is a comprehensive and well-written review of the field of hyperheuristics by Carsten Witt. The paper provides a thorough overview of the different types of hyperheuristics, their applications, and the challenges they face.

One of the strengths of the paper is its clear and concise writing style, which makes it accessible to a wide audience. Witt does an excellent job of explaining complex concepts in simple terms, making it easy for readers to understand the key ideas.

Another strength of the paper is its thorough coverage of the field. Witt covers a wide range of topics, including the history of hyperheuristics, their applications, and the challenges they face. He also provides numerous examples of how hyperheuristics have been used to solve real-world problems, which helps to illustrate the practical significance of the field.

One potential weakness of the paper is that it does not delve deeply into the technical details of the various hyperheuristics algorithms. While this makes the paper more accessible to a general audience, it may leave some readers wanting more information on how the algorithms work.

Overall, "Hyper-Heuristics: An Emerging Direction in Modern Search Technology" is a valuable resource for anyone interested in the field of hyperheuristics. It provides a comprehensive overview of the field and is written in a clear and concise style that makes it accessible to a wide audience.

"Hyper-Heuristics: An Emerging Direction in Modern Search Technology" by Carsten Witt is a comprehensive review of the field of hyperheuristics, which are high-level heuristics that guide the search process for other heuristics or metaheuristics. The paper begins by providing a background on the development of heuristics and metaheuristics and the challenges associated with their use in solving complex optimization problems.

Witt then introduces the concept of hyperheuristics and discusses the different types of hyperheuristics that have been proposed in the literature, including selection-based, generation-based, and learning-based approaches. He also discusses the advantages of using hyperheuristics, including the ability to adapt to changing problem conditions and the ability to handle multiple problem instances simultaneously.

The paper then goes on to review some of the key works in the field of hyperheuristics, including the work of Fukunaga and Muller on the use of machine learning techniques to guide the search process, the work of Burke et al. on the use of evolutionary algorithms to design hyperheuristics, and the work of Hyatt et al. on the use of swarm intelligence for hyperheuristic design.

Overall, Witt's review provides a thorough and informative overview of the field of hyperheuristics and its potential for addressing complex optimization problems. The paper is well-written and easy to follow, making it accessible to a wide audience. While the field of hyperheuristics is still in its early stages of development, the potential for this approach to significantly improve the performance of heuristics and metaheuristics makes it an exciting area of research that is worth further exploration.

Hyper-heuristics, as described in the survey paper "Hyper-Heuristics: A Survey of the State of the Art" by Edmund Burke and Graham Kendall, refer to high-level search strategies that aim to effectively select and/or generate low-level heuristics for use in solving computational problems. The authors describe the emergence of hyper-heuristics as a response to the growing complexity and diversity of optimization problems, as well as the increasing recognition of the limitations of traditional optimization methods.

The paper provides a comprehensive overview of the field, including its definitions, design principles, and various application areas. The authors also discuss the different types of hyper-heuristics, including adaptive, population-based, and portfolio-based approaches, and provide examples of each.

One of the strengths of the paper is its thorough coverage of the literature on hyper-heuristics. The authors review a wide range of papers and highlight the key contributions and limitations of each. They also provide a detailed taxonomy of the various methods proposed in the literature and discuss the challenges and future directions for research in the field.

Overall, the paper provides a valuable resource for researchers and practitioners interested in hyper-heuristics and offers a clear and concise overview of the state of the art in the field. However, one potential limitation is that the focus is mainly on theoretical aspects of hyper-heuristics, with less emphasis on practical implementation and case studies.

In summary, "Hyper-Heuristics: A Survey of the State of the Art" by Edmund Burke and Graham Kendall is a comprehensive and well-written survey of the field of hyper-heuristics, offering a valuable overview of the definitions, design principles, and application areas of these high-level search strategies.

Hyperheuristics have been an emerging area of research in modern search technology for the past few decades, with the goal of developing high-level strategies that can guide the search process for solving complex problems. In their review paper "Hyper-Heuristics: A Survey of the State of the Art," Edmund Burke and Graham Kendall provide an overview of the field, including its definitions, classification, and applications, as well as the various hyperheuristic methods that have been proposed and their performance.

The authors begin by defining hyperheuristics as high-level search strategies that operate on a set of low-level heuristics to guide the search process. They differentiate hyperheuristics from metaheuristics, which are high-level strategies that operate on a single low-level heuristic, and point out that hyperheuristics have the potential to improve the performance of metaheuristics by adaptively selecting the best heuristics for the given problem.

The authors then provide a classification of hyperheuristics based on the type of low-level heuristics they operate on, as well as their selection and adaptation mechanisms. They also discuss the various applications of hyperheuristics, including combinatorial optimization, scheduling, and automated planning.

In terms of the methods proposed for developing hyperheuristics, the authors discuss several approaches, including rule-based systems, machine learning, and evolutionary algorithms. They also provide a comparison of the performance of these methods on a range of benchmark problems.

Overall, the review by Burke and Kendall provides a comprehensive overview of the field of hyperheuristics, including its definitions, classification, and applications, as well as the various methods proposed for developing hyperheuristics and their performance. It is a valuable resource for researchers and practitioners interested in this area of modern search technology.

Hyperheuristics: An Emerging Direction in Modern Heuristics" by Edmund Burke is a comprehensive review of the current state of the field of hyperheuristics. Burke begins by defining hyperheuristics as a higher level heuristic that selects or generates lower level heuristics, and discusses the various ways in which this can be done.

One of the strengths of the review is its focus on the practical applications of hyperheuristics, including their use in real-world problems such as scheduling, vehicle routing, and protein folding. Burke also discusses the various evaluation methods used to measure the performance of hyperheuristics, including runtime, solution quality, and robustness.

One area where the review could be improved is in its discussion of the limitations of hyperheuristics. While Burke does mention some of the challenges faced by hyperheuristics, such as the difficulty in selecting appropriate low level heuristics and the need for a diverse set of candidate heuristics, he does not delve into these issues in great detail. It would also be helpful to have a more in-depth discussion of the relative strengths and weaknesses of different hyperheuristic approaches.

Overall, "Hyperheuristics: An Emerging Direction in Modern Heuristics" is a useful resource for researchers and practitioners interested in the field of hyperheuristics. It provides a clear overview of the current state of the field and highlights the potential of hyperheuristics for solving real-world problems.

Hyperheuristics are a relatively new field in optimization and search technology. They are a type of metaheuristic that focuses on selecting and adapting low-level heuristics in order to solve complex optimization problems. Hyperheuristics have gained increasing attention in recent years due to their ability to effectively deal with a wide range of optimization problems and their potential for parallel implementation.

The edited book "Hyperheuristics: An emerging direction in modern search technology" edited by Edmund Burke and Graham Kendall provides a comprehensive overview of the current state of the art in hyperheuristics research. The book is divided into four main sections, with each section covering a different aspect of hyperheuristics. The first section provides an introduction to hyperheuristics, including a historical overview and definitions of key terms. The second section covers different types of hyperheuristics, including population-based, trajectory-based, and hybrid approaches. The third section discusses applications of hyperheuristics in various domains, such as logistics, healthcare, and finance. The final section presents future directions and challenges in hyperheuristics research.

One of the key strengths of this book is the wide range of contributions from leading researchers in the field. The authors come from a variety of academic and industrial backgrounds, providing a diverse perspective on the current state and future developments of hyperheuristics. The book also includes several case studies, which provide practical examples of how hyperheuristics can be applied to real-world optimization problems.

Overall, "Hyperheuristics: An emerging direction in modern search technology" is a valuable resource for researchers and practitioners interested in hyperheuristics and optimization. It provides a comprehensive overview of the field, covering a wide range of topics and applications. The

contributions from leading researchers and the inclusion of case studies make it an essential reference for anyone working in this area.

Hyperheuristics, also known as high-level heuristics or meta-metaheuristics, refer to a class of search algorithms that aim to select and/or generate heuristics for specific problem instances. They are a relatively recent development in the field of optimization and search, and have been applied to a wide range of combinatorial optimization problems.

One of the key characteristics of hyperheuristics is their ability to adapt to different problem instances, making them potentially more widely applicable than traditional heuristics. This adaptability is typically achieved through the use of a selection mechanism, which chooses the most appropriate heuristic for a given problem, or a generation mechanism, which creates a new heuristic on the fly.

Some of the leading thinkers in the field of hyperheuristics include Edmund Burke, Andries Petrus Engelbrecht, Graham Kendall, Kenneth De Jong, and Peter Ross. Burke and Engelbrecht have edited several special issues and books on the topic, including the "Journal of Heuristics" and "Hyperheuristics: An emerging direction in modern search technology". Kendall and De Jong have also contributed significantly to the field, with their work on the "Hyper-Heuristics: A Survey of the State of the Art" providing a comprehensive overview of the current state of the field.

One of the key challenges in the development of hyperheuristics is the need to balance the exploration of new heuristics with the exploitation of known good ones. This trade-off, known as the exploration-exploitation dilemma, is a common theme in the literature on hyperheuristics. Other challenges include the development of robust and efficient selection and generation mechanisms, as well as the need to effectively evaluate and compare different hyperheuristic approaches.

There have been a number of successful applications of hyperheuristics to real-world problems, including scheduling, vehicle routing, and resource allocation. However, there is still much work to be done in the field, with many open questions and areas for future research.

PSPACE

PSPACE (Polynomial Space) is a complexity class in computational complexity theory. It is the set of all problems that can be solved by a deterministic Turing machine using a polynomial amount of space. In other words, these are the problems for which the amount of memory required to solve them is polynomial in the size of the input. PSPACE is a superset of the complexity class NP (nondeterministic polynomial time), which is the set of all problems for which a solution can be checked in polynomial time.

Some examples of problems that can be solved in PSPACE include Boolean satisfiability, linear programming, and the satisfiability of first-order logic formulas. PSPACE-complete problems are the hardest problems in PSPACE, meaning that any other problem in PSPACE can be reduced to a PSPACE-complete problem in polynomial time. The most well-known PSPACE-complete problem is the quantified Boolean formula problem.

Memetic Algorithms

Memetic algorithms are a type of hybrid optimization algorithm that combines elements of both metaheuristics and local search techniques. They are designed to explore the search space more effectively and efficiently by using a population-based approach to guide the search towards promising areas of the search space, while also using local search techniques to fine-tune and optimize solutions within those areas. Memetic algorithms have been successfully applied to a variety of optimization problems, including those in machine learning, image processing, and scheduling.

Post's Correspondence Problem

The Post Correspondence Problem (PCP) is a decision problem in which the task is to determine if there exists a sequence of words chosen from two given sets of words, such that the concatenation of the words is equal to a third given word. This problem is of interest in the field of theoretical computer science because it is an example of a problem that is solvable by a Turing machine but not solvable by a finite automaton. It was introduced by Emil Post in 1946 and is also known as the Post Correspondence Problem with Cards.

Polyomino Tiling's

A polyomino is a geometric shape formed by joining one or more equal-sized squares edge to edge. A polyomino tiling is a covering of a plane with non-overlapping polyominoes. The Post's Correspondence Problem (PCP) is a decision problem in computational complexity theory that asks whether it is possible to match the given collection of "dominoes" in the plane without overlaps and without gaps.

The PCP is a PSPACE-complete problem, meaning that it is a particularly difficult problem in the class of problems that can be solved using a reasonable amount of computer memory. This means that there is no known algorithm that can solve the PCP in polynomial time, and it is likely that no such algorithm exists. However, efficient algorithms do exist for solving certain special cases of the PCP, such as when the dominoes are all squares or when they are all rectangles.

Polyomino tiling has applications in areas such as image and pattern recognition, computer graphics, and error-correcting codes. It is also a subject of mathematical study in its own right, with connections to topics such as combinatorics and group theory.

Polyominoes have a long history, going back to the start of the 20th century, but they were popularized in the present era initially by **Solomon Golomb**, then by Martin Gardner in his Scientific American columns "Mathematical Games," and finally by many research papers by David Klarner.