Module 2: Algorithmic Strategies

Module overview

This module, module 2, covers the important techniques used in designing and analysing efficient algorithms. These algorithms include, among others, those of greedy algorithms, divide-and-conquer algorithms, dynamic programming algorithms, etc. The techniques may appear somewhat sophisticated, but we can see how they can be used to solve many computational problems.

Therefore, this second module delves into some algorithmic techniques. It discusses the concept of recursion as used in divide-and-conquer algorithms. While the use cases of the various strategies are discussed, this module provides the distinction amongst the strategies and offers ideas about the performance characteristics in terms of their efficiency using the asymptotic notations.

The units of this module present the algorithmic strategies in pairs most of the time, such as brute force algorithms and greedy algorithms in unit 1. As a matter of fact, this approach is not because of their relatedness, but for the reasons of the stipulations of the units of the modules of this course. However, the remaining units of this module also show such pairing of the various algorithmic strategies. Nevertheless, the respective strategies will be discussed by their separate merits. It is hoped that one can pick out the similarities and differences between and amongst the various strategies under consideration, and more importantly recognise how they can be used to solve computational problems of interest.

The module is broken down into five study units which are:

1. Study Unit 1: Brute Force; Greedy Algorithms
2. Study Unit 2: Divide and Conquer; Backtracking Algorithms
3. Study Unit 3: Dynamic Programming Algorithms
4. Pattern Matching Algorithms and Heuristics Algorithms
5. Dynamic Programming Algorithms

Study Unit 1: Brute Force and Greedy Algorithms

Expected Duration: # week

Introduction

In this unit, the topics of brute force algorithms and greedy algorithms will be treated under separate subunits. However, as far as optimization of functions may be concerned, the two strategies offer little assurances of producing optimal solutions in terms of efficiency. Algorithms for optimisation problems typically go through a sequence of steps with a set of

choices at each step. The usual and overriding objective is basically to choose the simpler and the more efficient algorithms.

Learning Outcome for Study Unit 1

At the end of this unit, you are expected to;

1.1 Define and use correctly the words in bold (SAQ1.1)

1.2 State the characteristics of brute force with illustrations (SAQ1.2)

1.3 State the characteristics of greedy algorithms with illustrations (SAQ1.3)

1.1 What does Brute Force Algorithm do?

Just as the name implies, a brute force algorithm solves a problem by exhaustively going through all the problem instances or choices until a solution is found. In other words, a brute force algorithm simply tries all possibilities until a satisfactory solution is found. The time complexity of a brute force algorithm is normally proportional to the input size and as such, brute force algorithms are usually simple and consistent, but can be quite slow especially when compared with say divide-and-conquer alternatives as input size increases without bound.

1.2 Characteristics of Brute force algorithm

A brute force algorithm can be:

(i) **Optimizing** - by finding the best solution. This may require finding all solutions (feasible solutions) or if a value for the best solution is known, it may stop when any best solution is found. (example: finding the best path for a travelling salesman.)

(ii) **Satisficing** - by stopping as soon as a solution is found that is good enough (example: finding a travelling salesman path that is within 10% of optimal.)

Notably, brute force approach is straightforward means to solving a problem without regard to efficiency.

1.2.1 Illustration of Brute force algorithm

Example: A **O**(n) algorithm for a^n:

algorithm Power (a, n)

// Input: A real number and an integer $n \geq 0$

// Output: a^n

result $\leftarrow$ 1

for i $\leftarrow$ 1 to n do

result $\leftarrow$ result * a

return result.

In-text Question:

A characteristic of the brute force algorithm that allows it to stop as soon as the solution is found is called?

Solution: Optimizing

Other examples include the insertion sort and sequential search algorithms discussed in module one, string matching, and other numeric computations, such as matrix multiplication, solving recurrences, etc. Despite the inefficiency of brute force algorithms, they are applicable to a wide range of problems, are simple to design, and may be useful in solving small problem instances.

1.3 Characteristics of Greedy Algorithms

A greedy algorithm is one that always makes the choice that looks best at a particular time or moment of an instance of a problem. This implies that a greedy algorithm makes a locally optimal choice in the hope that the choice made will eventually lead to a globally optimal solution. A greedy algorithm is more efficient than dynamic programming, for instance. It makes the choice that looks best at the moment. Though, greedy algorithms do not always yield optimal solutions, but they are effective for many problems.

Thus, we can assert that a greedy algorithm sometimes works well for optimization problems. A greedy algorithm works in phases.

At each phase,

i. You take the best you can get right now, without regard to future consequences.
ii. You hope that by choosing a local optimum at each step, you will end up at a global optimum.

1.3.1 Illustration of greedy algorithm

Examples:

(1) The problem of scheduling several competing activities that require exclusive use of a common resource, with a goal of selecting a maximum-size set of mutually compatible activities.

Suppose we have a set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed activities that wish to use a resource, such as a lecture hall, which can serve only one activity at a time. Each activity a_i has a start time s_i and a finish time f_i, where $0 \leq f_i < \infty$. If selected, activity a_i takes place during the half-open time interval $[s_i, f_i)$. Activities a_i and a_j are compatible if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. That is, a_i and a_j are compatible.

If $s_i \geq f_j$ or $s_j \geq f_i$.

In the activity – selection problem [cormen, et al, 2010, P415], it is required to select a maximum-size subject of mutually compatible activities. The assumption is that the activities are sorted in monotonically increasing order of finish time.

$$f_1 \leq f_2 \leq f_3 \leq \ldots \leq f_{n-1} \leq f_n$$

(2) Counting a certain amount of money, using the fewest possible bills and coins. A greedy algorithm would perform this task by talking the largest possible bill or coin that does not overshoot.

(3) Other greedy algorithms are;

(i) Dijkstra's algorithm for finding the shortest path in a graph: always takes the shortest edge connecting a known node to an unknown node.

(ii) Kruskal's algorithm for finding a minimum cost spanning tree: Always tries the lowest-cost remaining edge.

(iii) Prims algorithm for finding a minimum cost spanning tree: Always takes the lowest cost edge between nodes in the spanning tree and nodes not yet in the spanning tree, etc.

In-text Question:

A greedy algorithm is more efficient than dynamic programming because……?

Solution: It strives to consider several options and makes the best choice at each moment

1.4 Summary of Study Unit 1

In this unit, you have learned the following;

1. A brute force algorithm solves a problem by exhaustively going through all the problem instances or choices until a solution is found.
2. The time complexity of a brute force algorithm is normally proportional to the input
3. A brute force algorithm can be Optimizing and Satisficing
4. A greedy algorithm is one that always makes the choice that looks best at a particular time or moment of an instance of a problem.
5. Other greedy algorithms include Dijkstra's algorithmm, Kruskal's algorithm and Prims algorithm

1.5 Self-Assessment Questions (SAQs) for Study Unit 1

Now that you have completed this study session, you can check to see how well you have achieved its Learning Outcomes by answering the following questions.

SAQ 1.1 (Tests learning outcome1.1)

1. Define brute force algorithm

SAQ 1.2 (Tests learning outcome 1.2)

1. Mention two characteristics of Brute force algorithm

SAQ 1.3 (Tests learning outcome1.3)

1. One of the characteristics of a greedy algorithm is that it is more efficient than dynamic programming (TRUE/FALSE)?

References and Further Reading

i. Cormen T. H., Leiserson, C. E., Rivest, R. L., and Stein, C., Introduction to Algorithms, Massachusetts Institute of Technology, Fourth Edition, 2022.
ii. Sedgewick, R. and Wayne, K., Algorithms, Addison Wesley, Fourth Edition, 2011.
iii. Weiss, M. A., Bhattachatjee, A. K., and Mukherjee, S., Data Structures and Algorithms in Java, Pearson Education Limited, Third Edition, 2012
iv. Knuth, D. E., The Art of Computer Programming, Addison Wesley, Volumes 1 to 4, Third Edition, 2011
v. Heineman, G. T., Pollice G., and Selkow, S., Algorithms in a Nutshell, O'Reilly Media Inc., Second Edition, 2015
vi. And Numerous Internet Resources.

Study Unit 2: Divide-and-Conquer and Backtracking Algorithms

Expected Duration: # week

Introduction

In this unit, we shall explore two algorithms namely, divide-and-conquer and backtracking techniques, just as we did with the two discussed in unit 1 of this current module. So, we begin with divide-and-conquer algorithms and end the unit with backtracking algorithms.

Learning Outcome for Study Unit 2

At the end of this unit, you are expected to do the following.

2.1 Define and use correctly the words in bold (SAQ2.1)

2.2 List and describe the characteristics and examples of divide-and-conquer algorithms (SAQ 2.2)

2.3 List and describe the characteristics and examples of backtracking algorithms (SAQ 2.2)

2.1 What is Divide-and-conquer Algorithm?

The divide-and-conquer method is a powerful technique for designing efficient algorithms. This, for instance, means that the technique supports the development of solutions that are more asymptotically efficient as input sizes grow arbitrarily large. The divide-and-conquer technique often relies on the principle of recursion. **Recursion** is indispensable in divide-and-conquer technique and all others that are considerable for improving efficiency of computational methods.

2.2 Characteristics of Divide-and-conquer Algorithms

Typically, a divide-and-conquer algorithm consists of three characteristic steps.

(1) **A divide step**: that divides the problem into smaller subproblems of the same type and recursively solves the subproblems

(2) **A conquer step**: that conquers the subproblems by solving them recursively

(2) **A combine step**: That combines the solutions to the subproblems into a solution to the original problem.

Traditionally, an algorithm is only called divide-and-conquer if it contains two or more recursive calls. Essentially, a divide-and-conquer algorithm breaks down a large problem into smaller subproblems, which may even be down into smaller subproblems and so forth. The recursion is said to be bottom out when it reaches a base case and the subproblem is small enough to be solved directly without further recursing.

In-text Question:

Why is recursion indispensable to divide-and-conquer algorithms?

Solution: Recursion is important to divide-and-conquer algorithm because it makes room for thoroughness/exhaustiveness as it continues to go back to repeat an action.

2.2.1 Examples of divide-and-conquer algorithms:

(1) Quicksort:

a Partition the array into two parts, and quicksort each of the parts

b No additional work is required to combine the two sorted parts.

(2) Mergesort:

- Cut the array in half, and mergesort each half
- Combine the two sorted arrays into a single sorted array by merging them.

(3) Binary Search:

a Divide the array in two halves, and binary search each half for the target element.

b No additional work is required to combine the two searched halves.

2.3 Backtracking Algorithms

Backtracking is a general algorithm for finding all (or some) solutions to some computational problem that incrementally builds candidates to the solution, and abandons each partial candidate *c* ("backtracks") as soon as it determines that *c* cannot possibly be completed to a valid solution.

In other instances, say searching, backtracking is a systematic way to iterate through all the possible combinations of a search space. These configurations may represent all possible arrangements of objects (permutations) or all possible ways of building a collection of them (subsets). Other situations may demand enumerating all spanning trees of a graph, all paths between two vertices, or all possible ways to partition vertices into colour classes.

A backtracking algorithm is based on a **depth-first** recursive search. The algorithm proceeds as follows:

(1) Tests to see if a solution has been found, and if so, returns it; otherwise.

(2) For each choice that can be made at this point

- make that choice
- Recur
- If the recursion returns a solution, return it

(3) If no choices remain, return failure.

In-text Question:

The characteristic of divide-and-conquer algorithm that combines the solutions to the subproblems into a solution to the original problem is known as?

Solution: Combine step

2.3.1 Example of backtracking algorithm

- Colour a map with no more than 4 colours:

Colour (Country n)

(1) If all countries have been coloured (n > no of countries)

return success; otherwise,

(2) for each colour c of four colours,

(3) if country n is not adjacent to a country that has been coloured C

(4) - Colour country n with colour C

(5) - recursively colour country n + 1

(6) - If successful, return success

(7) Return failure (if loop exits)

2.4 Summary of Study Unit 2

In this unit, you have learned the following;

1. The divide-and-conquer method is a powerful technique for designing efficient algorithms.
2. The divide-and-conquer technique often relies on the principle of recursion
3. Three major characteristics of divide-and-conquer algorithm are: divide step, conquer step: and combine step:
4. Quick sort, merge sort and binary search are examples of divide-and-conquer algorithms.
5. A backtracking algorithm is based on a depth-first recursive search

2.5 Self-Assessment Questions (SAQs) for Study Unit 2

Now that you have completed this study session, you can check to see how well you have achieved its Learning Outcomes by answering the following questions.

SAQ 2.1 (Tests learning outcome2.1)

1. Define divide-and-conquer algorithm

SAQ 2.2 (Tests learning outcome 2.2)

1. Three major characteristics of divide-and-conquer algorithm are: ……, ……, and ……..

SAQ 2.3 (Tests learning outcome2.3)

1. Which of the following algorithms is associated with depth first?

A. Divide-and-conquer B. Backtracking

References and Further Reading

i. Cormen T. H., Leiserson, C. E., Rivest, R. L., and Stein, C., Introduction to Algorithms, Massachusetts Institute of Technology, Fourth Edition, 2022.

ii. Sedgewick, R. and Wayne, K., Algorithms, Addison Wesley, Fourth Edition, 2011.

iii. Weiss, M. A., Bhattachatjee, A. K., and Mukherjee, S., Data Structures and Algorithms in Java, Pearson Education Limited, Third Edition, 2012

iv. Knuth, D. E., The Art of Computer Programming, Addison Wesley, Volumes 1 to 4, Third Edition, 2011

v. Heineman, G. T., Pollice G., and Selkow, S., Algorithms in a Nutshell, O'Reilly Media Inc., Second Edition, 2015

vi. And Numerous Internet Resources.

Study Unit 3: Dynamic Programming Algorithms

Expected Duration: # week

Introduction

The name **dynamic programming**, used in the context of algorithms, refers more to the use of a **tabular method**, and not necessarily to writing computer code. Just like divide-and-conquer technique, dynamic programming solves problems by combining the solutions to subproblems. We recall that divide-and-conquer algorithms partition a problem, in the divide step, into disjoint subproblems, conquers by solving the subproblems recursively, and then combines their solutions to solve the original problem. On the part of dynamic programming, we see that the technique applies when the subproblems overlap; which implies when subproblems share subproblems.

In contrast, therefore, whilst a divide-and-conquer algorithm does more work than necessary, by repeatedly solving the common subproblems, a dynamic programming algorithm solves each subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subproblem. The emphasis of this unit is on dynamic programming.

Learning Outcome for Study Unit 3

At the end of this unit, you are expected to do the following;

3.1 Define and use correctly the words in bold (SAQ 3.1)

3.2 List and explain the steps involved in dynamic programming (SAQ 3.2)

3.3 Enumerate some applications of dynamic programming algorithms (SAQ 3.3)

3.1 Characteristics of Dynamic Programming Algorithms

Dynamic programming is certainly a method of solving complex problems by breaking them down to simpler subproblems. The method is applicable to problems exhibiting the properties of **overlapping subproblems** and **optimal substructure**. Dynamic programming typically applies to **optimisation** problems in which there are many possible or feasible solutions.

Each solution has a value, and the objective is usually to find a solution with the optimal (minimum or maximum) value.

Dynamic programming technique is more efficient than other methods that do not take advantage of the subrountine overlap (like depth-first search). The approach is especially useful when the number of repeating subproblems grows exponentially as a function of the input size. The approach seeks to solve each subproblem only once, thus reducing the number of computations.
In effect, a dynamic programming algorithm remembers past results (memo-ized) and uses them to find new results. Dynamic programming is generally used for optimization problems in which the following apply;

(i) Multiple solutions exist: need to find the best one
(ii) Requires optimal substructure and overlapping subproblem
(iii) Optimal substructure: Optimal solution contains optimal solutions to subproblemss
(iv) Overlapping subproblems: solutions to subproblems can be stored and re-used in a bottom-up fashion.

3.2 Steps involved in dynamic programming

Dynamic programming differs from divide-and-conquer as the subproblems generally need not overlap-divide-and-conquer. To develop a dynamic programming algorithm, the following sequence of four steps can be followed;

a. Characterise the structure of an optimal solution.
b. Recursively define the value of an optimal solution.
c. Compute the value of an optimal solution, typically in a bottom-up fashion.
d. Construct an optimal solution from computed information.

It is noteworthy that steps *a* to *c* form the basis of a dynamic programming solution to a problem. If only the value of an optimal solution is required, and not the solution is itself, then one can omit step *d*. For example, in a problem requiring the determination of the number of medical teams to allocate to various countries to achieve maximum life expectancy, the value of the optimal solution is the highest total value of life expectancy

achieved across the countries under consideration, while the optimal solution is the number of teams assigned per country to achieve that value of optimal solution.

In-text Question:

Dynamic programming is generally used for optimization problems, TURE/FALSE?

Solution: TRUE

3.3 Examples of Dynamic Programming Algorithms:

Several examples exist, even in literature, as there is no standard formulation for DPPs. Some are'

(1) Rod-cutting: Deciding where to cut steel rods; where an enterprise wishes to know the best way to cut up the rods (Cormen, et al, 2010, page 360).

Problem: Given a rod of length n inches and a table of prizes P_i for i = 1, 2, …, n. Determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. Note that if the prize P_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

(2) Some example in bioinformatics, such as

(a) Compute an optimal pairwise alignment.

- Optimal substructure: the alignment of two prefixes contains solutions for the optimal alignments of smaller prefixes.

- Overlapping subproblems: The solution for the optimal alignment of two prefixes can be constructed using the stored solutions of the alignment of three subproblems (in a linear gap model).

(b) Compute a Viterbi path in HMM

- Optimal substructure: The viterbi path for an input prefix ending in a state of an HMM (Hidden Markov Model) contains shorter Viterbi paths for smaller parts of the input and other HMM states.

- Overlapping subproblems: The solution for the Viterbi path for an input prefix ending in a state of an HMM can be constructed using the stored solutions of Viterbi paths for a shorter input prefix and all HMM states.

In-text Question:

What is the meaning of overlapping subroutines?

Solution: It refers to subroutines that depend on other subroutines for input.

3.4 Summary of Study Unit 3

In this unit, you have learned the following;

1. Dynamic programming is a method of solving complex problems by breaking them down to simpler subproblems.
2. Dynamic programming typically applies to optimisation problems in which there are many possible or feasible solutions
3. The objective of dynamic programming is to find solution with the optimal (minimum or maximum) value.
4. Dynamic programming is generally used for optimization problems.
5. The four major steps in dynamic programming include:

   a. Characterise the structure of an optimal solution.
   b. Recursively define the value of an optimal solution.
   c. Compute the value of an optimal solution, typically in a bottom-up fashion.
   d. Construct an optimal solution from computed information.

6. Dynamic programming can be applied in rod-cutting and bioinformatics.

3.5 Self-Assessment Questions (SAQs) for Study Unit 3

Now that you have completed this study session, you can check to see how well you have achieved its Learning Outcomes by answering the following questions.

SAQ 3.1 (Tests learning outcome 3.1)

1. What is dynamic programming?

SAQ 3.2 (Tests learning outcome 3.2)

1. Arrange the following dynamic programming steps in their logical order:

   a. Construct an optimal solution from computed information.
   b. Compute the value of an optimal solution, typically in a bottom-up fashion..
   c. Recursively define the value of an optimal solution
   d. Characterise the structure of an optimal solution

SAQ 3.3 (Tests learning outcome 3.3)

1. Mention two application areas of dynamic programming

References and Further Reading

i. Cormen T. H., Leiserson, C. E., Rivest, R. L., and Stein, C., Introduction to Algorithms, Massachusetts Institute of Technology, Fourth Edition, 2022.
ii. Sedgewick, R. and Wayne, K., Algorithms, Addison Wesley, Fourth Edition, 2011.
iii. Weiss, M. A., Bhattachatjee, A. K., and Mukherjee, S., Data Structures and Algorithms in Java, Pearson Education Limited, Third Edition, 2012
iv. Knuth, D. E., The Art of Computer Programming, Addison Wesley, Volumes 1 to 4, Third Edition, 2011
v. Heineman, G. T., Pollice G., and Selkow, S., Algorithms in a Nutshell, O'Reilly Media Inc., Second Edition, 2015
vi. And Numerous Internet Resources.

Study Unit 4: Pattern Matching, and Heuristics Algorithms

Expected Duration: # week

Introduction

In this unit, the topics of pattern matching algorithms and heuristic algorithms will be treated under separate subunits. However, while pattern matching deals with assistive search function, heuristics algorithms provide efficient support towards arriving at near optimal solutions to complex problems where it may be difficult, if not impossible, to empirically achieve the exact optimal solution to complex optimisation problems. Thus, the unit is rendered respectively in terms of pattern matching algorithms and heuristic algorithms.

Learning Outcome for Study Unit 4

At the end of this unit, you are expected to;

4.1 Define and use correctly the words in bold (SAQ 4.1)

4.2 State the characteristics of pattern matching algorithms (SAQ 4.2)

4.3 Explain the meaning of heuristic algorithms (SAQ 4.3)

4.4 State some popular heuristic algorithms (SAQ 4.4)

4.5 Applications of heuristic algorithms (SAQ 4.5)

4.1 What is Pattern Matching Algorithm?

Pattern matching is a term often used interchangeably with string matching without loss of generality. **Text-editing programs**, in other words text-processing systems, frequently need to find all occurrences of a pattern in the text. Precisely, the systems must allow their users to search for a given character string within a body of text.

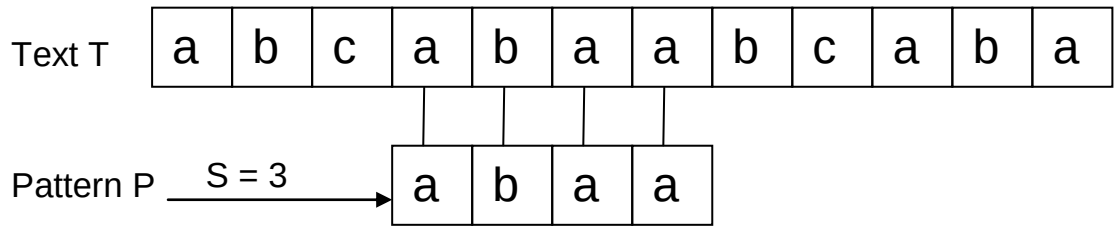
For the purposes of database management systems, the systems must be capable of searching for records with stated values in specified fields. Such problems typically pose the following string-matching problem: For a specified set {X(i), Y(i)} of pairs of strings, determine, if possible, an r such that X(r) = Y(r). Usually, the set is specified not by explicit enumeration of the pairs, but rather by a rule for computing the pairs <X(i), Y(i)> from some given data. There are several algorithms, especially **randomized algorithm,** for solving problems of this nature, such as those of (Karp and Rabin, 1987). There are many other applications for which

string-matching is known to be very essential. Some of which are searching for particular patterns in DNA sequences, finding web pages that are relevant to queries by Internet search engines, etc.

4.2 Characteristics of pattern matching algorithm

According to (Cormen, et al, 2022) we can formalize a string-matching problem as follows: assuming a text to be an array T[1…n] of length n and that the pattern is an array P[1…m] of length $m \leq n$. With a further assumption that the elements of P and T are characters drawn from a finite **alphabet** Σ. For example, we may have $\Sigma = \{0,1\}$ or $\Sigma = \{a, b, \ldots, z\}$. The character arrays P and T are often referred to as strings of characters.

Considering the following figure:



As indicated in the above string-matching problem, the pattern P can be said to occur with shift S in the text T. Equivalently, the pattern P occurs beginning at position S+1 in text T. This implies that $0 \leq S \leq n-m$ and $T[S+1, \ldots S+m] = P[1 .. m]$, that is $T[s+j] = P[j]$, for $1 \leq j \leq m$. If P occurs with shift S in T, then we call S a valid shift; otherwise, we call S an invalid shift. The string-matching problem is the problem of finding all valid shifts, with which a given pattern occurs in a given text T.

There are several proposals for pattern-matching algorithms, which progressively address the issue of improved performance. These include those of Radin-Karp, Knuth-Morris-Pratt, Bayer-Moore (BM), Horspool (HORSPOOL), Raita (RAITA), (Sheik, et al, 2003).

Some analytical results of the comparative study of performance of some of the algorithms are shown in the following table:

Algorithm	Pre-processing Time	Matching Time
Naïve	0	0((n-m+1)m)
Rabin-Karp	0(m)	0((n-m+1)m)
Finite automaton	0(m $\lvert\Sigma\rvert$)	0(n)
Knuth-Morris-Pratt	0(m)	0(n)

However, we limit our discussion to the Naïve algorithm. The naïve algorithm represents the direct way of doing the business of string-matching. It is also referred to as the naïve brute-force algorithm. Being a brute-force method, it operates by comparing the first m-characters of the text and the pattern in some predefined order and, after a match or a mismatch; it slides the entire pattern by one character in the forward direction of the text.

The above process is repeated until the pattern is positioned at the (n-m+1) position of the text.

NB: Recall that it is for this same task that several algorithms have been proposed, and these have their own advantages and limitations based on the pattern length, periodicity, and the type of text (for example, nucleotide or amino acid sequences or language characters, etc).

Next, we present the naïve string-matching algorithm, which finds all valid shift, using a loop that checks the condition P[1..m] = T[s+1..s+m] for each of the n-m+1 possible values of s.

NAÏVE-STRING-MATCHER (T, P)

1. n = T. length
2. m = P. length
3. for S = 0 to n-m
4. if P[1..m] = = T [s+1 .. s+m]
5. print “Pattern occurs with shift” S

Comments:

- The “for” loop of lines 3 – 5 considers each possible shift explicitly.

- The text in line 4 determines whether the current shift is valid; this test implicitly loops to check corresponding character positions until all positions match successfully or a mismatch is found.

4.3 The meaning of heuristics algorithms

Several authors and researchers are working in the field of heuristic algorithms with some notable achievements and application areas for this remarkable time- and cost-saving algorithmic strategy. In mathematical programming, a heuristic algorithm is a procedure that determines near-optimal solutions to an optimization problem. However, this is achieved by trading-in optimality, completeness, accuracy, or precision for speed. Nevertheless, heuristics is a widely used technique for a variety of reasons:

- Problems that do not have an exact solution or for which the formulation is unknown
- The computation of a problem is computationally intensive
- Calculation of bounds on the optimal solution in branch and bound solution processes

4.3.1 Methodology

Optimization heuristics can be categorized into two broad classes depending on the way the solution domain is organized:

a. Construction methods (Greedy algorithms)

The greedy algorithm works in phases, where the algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem. It is a technique used to solve the famous “travelling salesman problem” where the heuristic followed is: "At each step of the journey, visit the nearest unvisited city."

Example: Scheduling Problem

You are given a set of N schedules of lectures for a single day at a university. The schedule for a specific lecture is of the form (s time, f time) where s time represents the start time for that lecture, and similarly, the f time represents the finishing time. Given a list of N lecture schedules, we need to select a maximum set of lectures to be held out during the day such that none of the lectures overlaps with one another i.e. if lecture L_i and L_j are included in our

selection then the start time of j ≥ finish time of i or vice versa. The most optimal solution to this would be to consider the earliest finishing time first. We would sort the intervals according to the increasing order of their finishing times and then start selecting intervals from the very beginning.

b. Local Search methods

The Local Search method follows an iterative approach where we start with some initial solution, explore the neighbourhood of the current solution, and then replace the current solution with a better solution. For this method, the "travelling salesman problem" would follow the heuristic in which a solution is a cycle containing all nodes of the graph and the target is to minimize the total length of the cycle.

Example Problem

Suppose that the problem P is to find an optimal ordering of N jobs in a manufacturing system. A solution to this problem can be described as an N-vector of job numbers, in which the position of each job in the vector defines the order in which the job will be processed. For example, [3, 4, 1, 6, 5, 2] is a possible ordering of 6 jobs, where job 3 is processed first, followed by job 4, then job 1, and so on, finishing with job 2. Define now M as the set of moves that produce new orderings by the swapping of any two jobs. For example, [3, 1, 4, 6, 5, 2] is obtained by swapping the positions of jobs 4 and 1.

In-text Question:

The two broad categories of heuristic algorithm are ?

Solution: Construction methods and Local search methods

4.4 Popular Heuristic Algorithms

a. Genetic Algorithm

The term Genetic Algorithm was first used by John Holland. They are designed to mimic the Darwinian theory of evolution, which states that populations of species evolve to produce more complex organisms and fitter for survival on Earth. Genetic algorithms operate on string structures, like biological structures, which are evolving in time according to the rule of survival of the fittest by using a randomized yet structured information exchange. Thus, in every generation, a new set of strings is created, using parts of the fittest members of the old

set. The algorithm terminates when the satisfactory fitness level has been reached for the population or the maximum generations have been reached. The typical steps are:

1. Choose an initial population of candidate solutions
2. Calculate the fitness, how well the solution is, of each individual
3. Perform crossover from the population. The operation is to randomly choose some pair of individuals like parents and exchange some parts from the parents to generate new individuals
4. Mutation is to randomly change some individuals to create other new individuals
5. Evaluate the fitness of the offspring
6. Select the survive individuals
7. Proceed from 3 if the termination criteria have not been reached

b. Tabu Search Algorithm

Tabu Search (TS) is a heuristic algorithm created by Fred Glover using a gradient-descent search with memory techniques to avoid cycling for determining an optimal solution. It does so by forbidding or penalizing moves that take the solution, in the next iteration, to points in the solution space previously visited. The algorithm spends some memory to keep a Tabu list of forbidden moves, which are the moves of the previous iterations or moves that might be considered unwanted. A general algorithm is as follows:

1. Select an initial solution $s_0 \in S$. Initialize the Tabu List $L_0 = \emptyset$ and select a list tabu size. Establish $k = 0$.
2. Determine the neighbourhood feasibility $N(s_k)$ that excludes inferior members of the tabu list L_k.
3. Select the next movement s_{k+1} from $N(S_k)$ or L_k if there is a better solution and update L_{k+1}
4. Stop if a condition of termination is reached, else, $k = k + 1$ and return to 1

Example: The Classical Vehicle Routing Problem

Vehicle Routing Problems have very important applications in distribution management and have become some of the most studied problems in the combinatorial optimization literature. These include several Tabu Search implementations that currently rank among the most effective. The *Classical Vehicle Routing Problem* (CVRP) is the basic variant in that class of problems. It can formally be defined as follows. Let $G = (V, A)$ be a graph where V is the vertex set and A is the arc set. One of the vertices represents the *depot* at which a fleet of identical vehicles of capacity Q is based, and the other vertices customers that need to be serviced. With each customer vertex v_i are associated a demand q_i and a service time t_i. With each arc (v_i, v_j) of A are associated a cost c_{ij} and a travel time t_{ij}. The CVRP consists of finding a set of routes such that:

1. Each route begins and ends at the depot
2. Each customer is visited exactly once by exactly one route
3. The total demand of the customers assigned to each route does not exceed Q
4. The total duration of each route (including travel and service times) does not exceed a specified value L
5. The total cost of the routes is minimized

A feasible solution for the problem thus consists of a partition of the customers into m groups, each of total demand no larger than Q, that are sequenced to yield routes (starting and ending at the depot) of duration no larger than L.

c. Simulated Annealing Algorithm

The Simulated Annealing Algorithm was developed by Kirkpatrick et. al. in 1983 and is based on the analogy of ideal crystals in thermodynamics. The annealing process in metallurgy can make particles arrange themselves in the position with minima potential as the temperature is slowly decreased. The Simulation Annealing algorithm mimics this mechanism and uses the objective function of an optimization problem instead of the energy of a material to arrive at a solution. A general algorithm is as follows:

1. Fix initial temperature (T^0)

2. Generate starting point $\mathbf{x}^0$ (this is the best point $\boldsymbol{X}^*$ at present)

3. Generate randomly point $\boldsymbol{X}^S$ (neighboring point)

4. Accept $\boldsymbol{X}^S$ as $\boldsymbol{X}^*$ (currently best solution) if an acceptance criterion is met. This must be such a condition that the probability of accepting a worse point is greater than zero, particularly at higher temperatures

5. If an equilibrium condition is satisfied, go to (6), otherwise jump back to (3).

6. If termination conditions are not met, decrease the temperature according to a certain cooling scheme and jump back to (1). If the termination conditions are satisfied, stop calculations accepting the current best value $\boldsymbol{X}^*$ as the final ('optimal') solution.

Numerical Example: Knapsack Problem

One of the most common applications of the heuristic algorithm is the Knapsack Problem, in which a given set of items (each with a mass and a value) are grouped to have a maximum value while being under a certain mass limit. It uses the Greedy Approximation Algorithm to sort the items based on their value per unit mass and then includes the items with the highest value per unit mass if there is still space remaining.

Example

The following table specifies the weights and values per unit of five different products held in storage. The quantity of each product is unlimited. A plane with a weight capacity of 13 is to be used, for one trip only, to transport the products. We would like to know how many units of each product should be loaded onto the plane to maximize the value of goods shipped.

Product (i)	**Weight per unit (w_i)**	**Value per unit (v_i)**
1	7	9
2	5	4
3	4	3
4	3	2
5	1	0.5

Solution:

(a) Stages:

We view each type of product as a stage, so there are 5 stages. We can also add a sixth stage representing the endpoint after deciding

(b) States:

We can view the remaining capacity as states, so there are 14 states in each stage: 0,1, 2, 3, …13

(c) Possible decisions at each stage:

Suppose we are in state s in stage n (n < 6), hence there are s capacity remaining. Then the possible number of items we can pack is:

$j = 0, 1, \ldots [s/w_n]$

For each such action j, we can have an arc going from the state s in stage n to the state $n – j*w_n$ in stage n + 1. For each arc in the graph, there is a corresponding benefit $j*v_n$. We are trying to find a maximum benefit path from state 13 in stage 1, to stage 6.

(d) Optimization function:

Let $f_n(s)$ be the value of the maximum benefit possible with items of type n or greater using total capacity at most s

(e) Boundary conditions:

The sixth stage should have all zeros, that is, $f_6(s) = 0$ for each s = 0,1, … 13

(f) Recurrence relation:

$f_n(s) = \max \{j*v_n + f_{n+1}(s – j*w_n)\}, j = 0, 1, \ldots, [s/w_n]$

(g) Compute:

<u>In-text Question:</u>

Why is the Knapsack problem regarded as optimization problem?

<u>Solution</u>: This is because a given set of items (each with a mass and a value) are grouped to have a maximum value while being under a certain mass limit.

The solution will not show all the computations steps. Instead, only a few cases are given below to illustrate the idea.

- For stage 5, $f_5(s) = \max_{j=0, 1, \ldots [s/1]} \{j*0.5 + 0\} = 0.5s$ because given the all zero states in stage 6, the maximum possible value is to use up all the remaining s capacity.

- For stage 4, state 7,

$f_4(7) = \max_{j=0,1,\ldots,[7/w4]} = \{j*v_4 + f_5(7 - w_{4*}j)\}$

= max {0 + 3.5; 2 + 2; 4 + 0.5}

= 4.5

Using the recurrence relation above, we get the following table:

Unused Capacity	**Type 1**		**Type 2**		**Type 3**		**Type 4**		**Type 5**		
s	**$f_1(s)$**	**opt**	**$f_2(s)$**	**opt**	**$f_3(s)$**	**opt**	**$f_4(s)$**	**opt**	**$f_5(s)$**	**opt**	**$f_6(s)$**
13	13.5	1	10	2	9.5	3	8.5	4	6.5	13	0
12	13	1	9	2	9	3	8	4	6	12	0
11	12	1	8.5	2	8	2	7	3	5.5	11	0
10	11	1	8	2	7	2	6.5	3	5	10	0
9	10	1	7	1	6.5	2	6	3	4.5	9	0
8	9.5	1	6	1	6	2	5	2	4	8	0
7	9	1	5	1	5	1	4.5	2	3.5	7	0
6	4.5	0	4.5	1	4	1	4	2	3	6	0
5	4	0	4	1	3.5	1	3	1	2.5	5	0
4	3	0	3	0	3	1	2.5	1	2	4	0
3	2	0	2	0	2	0	2	1	1.5	3	0
2	1	0	1	0	1	0	1	0	1	2	0
1	0.5	0	0.5	0	0.5	0	0.5	0	0.5	1	0
0	0	0	0	0	0	0	0	0	0	0	0

Optimal solution: The maximum benefit possible is 13.5. Tracing forward to get the optimal solution: the optimal decision corresponding to the entry 13.5 for $f_1(1)$ is 1, therefore we should pack 1 unit of type 1. After that we have 6 capacity remaining, so look at $f_2(6)$ which is 4.5, corresponding to the optimal decision of packing 1 unit of type 2. After this, we have 6-5 = 1 capacity remaining, and $f_3(1) = f_4(1) = 0$, which means we are not able to pack any

type 3 or type 4. Hence we go to stage 5 and find that $f_5(1) = 1$, so we should pack 1 unit of type 5. This gives the entire optimal solution as can be seen in the table below:

Optimal solution

Product (i) Number of units

1 1

2 1

5 1

4.5 Applications of heuristic algorithms

Heuristic algorithms have become an important technique in solving current real-world problems. Its applications can range from optimizing the power flow in modern power systems to groundwater pumping simulation models. Heuristic optimization techniques are increasingly applied in environmental engineering applications as well such as the design of a multilayer sorptive barrier system for landfill liner. Heuristic algorithms have also been applied in the fields of bioinformatics, computational biology, and systems biology.

In-text Question:

One major aspect of life where heuristic algorithm is applied is?

Solution: Environmental engineering

4.4 Summary of Study Unit 4

In this unit, you have learned the following;

1. Pattern matching is a term often used interchangeably with string matching without loss of generality.
2. A heuristic algorithm is a procedure that determines near-optimal solutions to an optimization problem
3. Optimization heuristics can be categorized into two broad classes which are::
   a. Construction methods b. Local search methods
4. Genetic algorithm, Tabo search, and Simulated Annealing are all typical heuristic algorithms.

5. Major areas of heuristic algorithm application include; bioinformatics, computational biology, and systems biology

4.5 Self-Assessment Questions (SAQs) for Study Unit 4

Now that you have completed this study session, you can check to see how well you have achieved its Learning Outcomes by answering the following questions.

SAQ 4.1 (Tests learning outcome 4.1)

1. The term pattern matching can be said to mean the same as string matching (TRUE/FALSE)?

SAQ 4.2 (Tests learning outcome 4.2)

1. A major characteristic of pattern matching is that characters are drawn from an infinite alphabet. (TRUE/FALSE)?

SAQ 4.3 (Tests learning outcome 4.3)

1. A procedure that determines near-optimal solutions to an optimization problem is called?

SAQ 4.4 (Tests learning outcome 4.4)

1. Three popular heuristic algorithms include;

SAQ 4.5 (Tests learning outcome 4.5)

1. Apart from environmental engineering, mention two other areas of heuristic algorithm application

References for further reading

Eiselt, Horst A et al. Integer Programming and Network Models. Springer, 2011.

Introduction to Algorithms (Cormen, Leiserson, Rivest, and Stein) 2001, Chapter 16 "Greedy Algorithms".

Eiselt, Horst A et al. Integer Programming and Network Models. Springer, 2011.

J.H. Holland (1975) *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, Michigan; re-issued by MIT Press (1992).

Optimal design of heat exchanger networks, Editor(s): Wilfried Roetzel, Xing Luo, Dezhen Chen, Design and Operation of Heat Exchangers and their Networks, Academic Press, 2020, Pages 231-317, ISBN 9780128178942, https://doi.org/10.1016/B978-0-12-817894-2.00006-6.

Wang FS., Chen LH. (2013) Genetic Algorithms. In: Dubitzky W., Wolkenhauer O., Cho KH., Yokota H. (eds) Encyclopedia of Systems Biology. Springer, New York, NY. https://doi.org/10.1007/978-1-4419-9863-7_412

Fred Glover (1986). "Future Paths for Integer Programming and Links to Artificial Intelligence". Computers and Operations Research. **13** (5): 533–549,https://doi.org/10.1016/0305-0548(86)90048-1

Optimization of Preventive Maintenance Program for Imaging Equipment in Hospitals, Editor(s): Zdravko Kravanja, Miloš Bogataj, Computer-Aided Chemical Engineering, Elsevier, Volume 38, 2016, Pages 1833-1838, ISSN 1570-7946, ISBN 9780444634283, https://doi.org/10.1016/B978-0-444-63428-3.50310-6.

Glover, Fred, and Gary A Kochenberger. Handbook Of Metaheuristics. Kluwer Academic Publishers, 2003.

Kirkpatrick, S., Gelatt, C., & Vecchi, M. (1983). Optimization by Simulated Annealing. *Science*, *220*(4598), 671-680. Retrieved November 25, 2020, from http://www.jstor.org/stable/1690046

Brief review of static optimization methods, Editor(s): Stanisław Sieniutycz, Jacek Jeżowski, Energy Optimization in Process Systems and Fuel Cells (Third Edition), Elsevier, 2018, Pages 1-41, ISBN 9780081025574, https://doi.org/10.1016/B978-0-08-102557-4.00001-3.

NIU, M., WAN, C. & Xu, Z. A review on applications of heuristic optimization algorithms for optimal power flow in modern power systems. J. Mod. Power Syst. Clean Energy 2, 289–297 (2014), https://doi.org/10.1007/s40565-014-0089-4

J. L. Wang, Y. H. Lin and M. D. Lin, "Application of heuristic algorithms on groundwater pumping source identification problems," 2015 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM), Singapore, 2015, pp. 858-862, https://doi.org/10.1109/IEEM.2015.7385770.

Matott, L. Shawn, et al. "Application of Heuristic Optimization Techniques and Algorithm Tuning to Multilayered Sorptive Barrier Design." Environmental Science & Technology, vol. 40, no. 20, 2006, pp. 6354–6360., https://doi.org/10.1021/es052560+.

Larranaga P, Calvo B, Santana R, Bielza C, Galdiano J, Inza I, Lozano JA, Armananzas R, Santafe G, Perez A, Robles V (2006) Machine learning in bioinformatics. Brief Bioinform 7(1):86–112

Notes to the Self- Assessment Questions (SAQs) for Study Unit 1

SAQ 1.1 (Tests learning outcome 1.1)

1. A brute force algorithm solves a problem by exhaustively going through all the problem instances or choices until a solution is found

SAQ 1.2 (Tests learning outcome 1.2)

1. Two characteristics of brute force algorithm are Optimizing and Satisficing

SAQ 1.3 (Tests learning outcome 1.3)

1. TRUE

Notes to the Self- Assessment Questions (SAQs) for Study Unit 2

SAQ 2.1 (Tests learning outcome 2.1)

1. The divide-and-conquer method is a powerful technique for designing efficient algorithms.

SAQ 2.2 (Tests learning outcome 2.2)

1. Three major characteristics of divide-and-conquer algorithm are: divide step, conquer step: and combine step

SAQ 2.3 (Tests learning outcome 2.3)

1. Backtracking

Notes to the Self- Assessment Questions (SAQs) for Study Unit 3

SAQ 3.1 (Tests learning outcome 3.1)

1. Dynamic programming is simply a method of solving complex problems by breaking them down to simpler subproblems

SAQ 3.2 (Tests learning outcome 3.2)

1. The right order is D-C-B-A

SAQ 3.3 (Tests learning outcome 3.3)

1. Two application areas of dynamic programming are Rod cutting and bioinformatics

Notes to the Self- Assessment Questions (SAQs) for Study Unit 4

SAQ 4.1 (Tests learning outcome 4.1)

1. TRUE

SAQ 4.2 (Tests learning outcome 4.2)

1. FALSE

SAQ 4.3 (Tests learning outcome 4.3)

1. Heuristic algorithm

SAQ 4.4 (Tests learning outcome 4.4)

1. Popular heuristic algorithms include; Genetic algorithm, Tabo search, and Simulated Annealing algorithms.

SAQ 4.5 (Tests learning outcome 4.5)

1. Two other applications of heuristic algorithm are; bioinformatics and computational biology