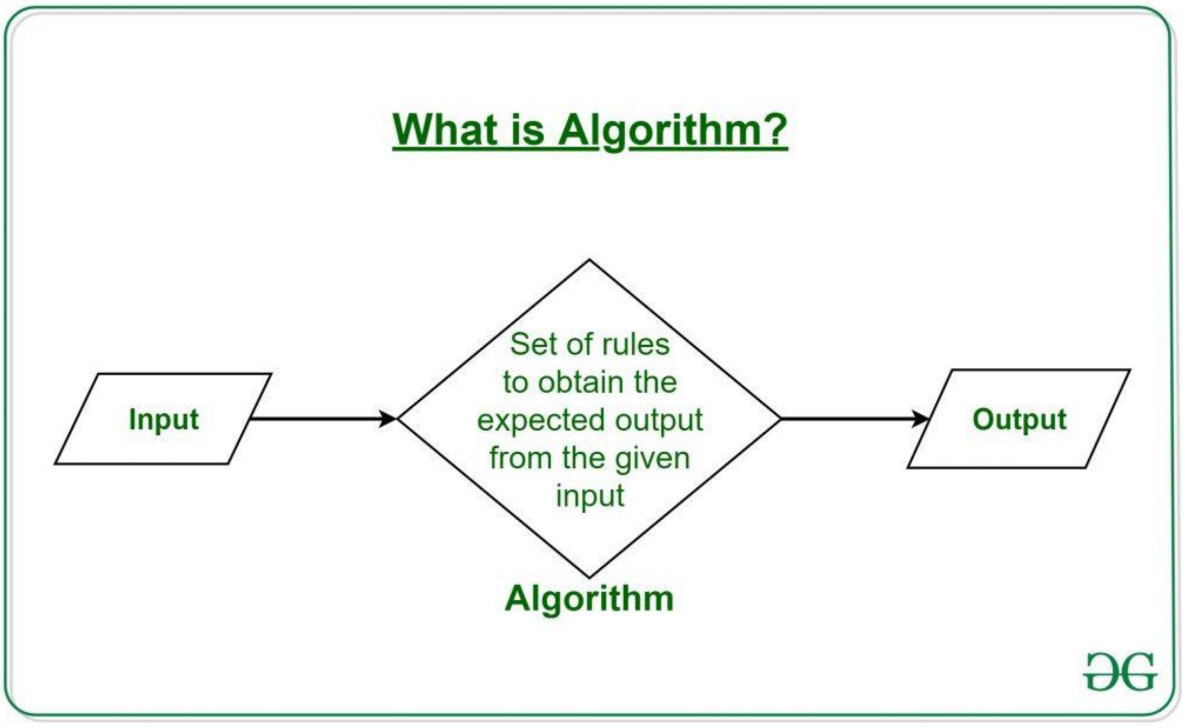
PATHFINDING

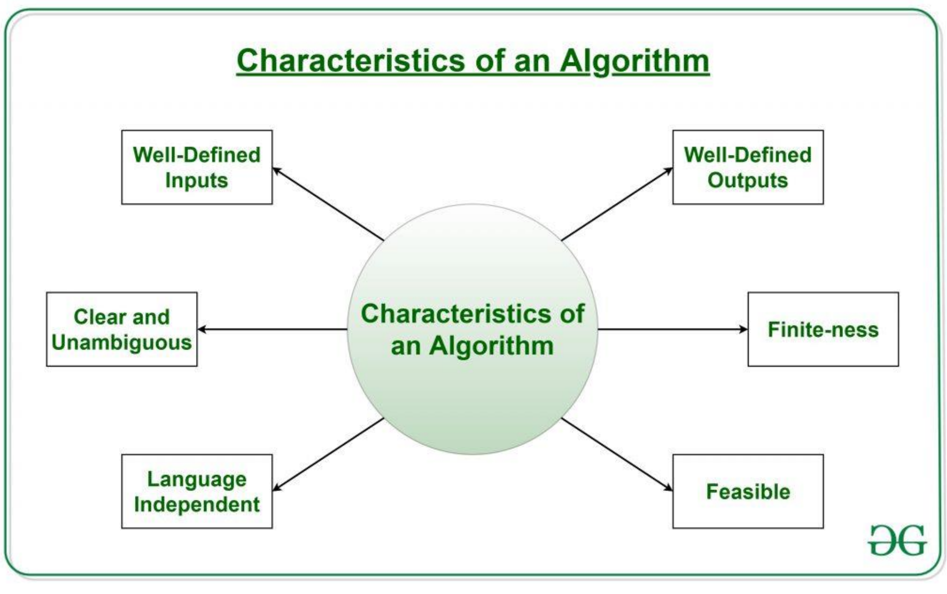
ALGORITHM VISUALIZER

What is an Algorithm?

The word Algorithm means " A set of finite rules or instructions to be followed in calculations or other problem-solving operations " Or " A procedure for solving a mathematical problem in a finite number of steps that frequently involves recursive operations".


What is Algorithm?
Input
Set of rules to obtain the expected output from the given input
Output
Algorithm

What are the Characteristics of an Algorithm?



In order for some instructions to be an algorithm, it must have the following characteristics:

- Clear and Unambiguous: The algorithm should be clear and unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- Well-Defined Inputs: If an algorithm says to take inputs, it should be well-defined inputs. It may or may not take input.
- Well-Defined Outputs: The algorithm must clearly define what output will be yielded and it should be well-defined as well. It should take at least 1 output.
- Finite-ness: The algorithm must be finite, i.e. it should terminate after a finite time.
- Feasible: The algorithm must be simple, generic, and practical, such that it can be executed with the available resources. It must not contain some future technology or anything.
- Language Independent: The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.

Properties of Algorithm:

- It should terminate after a finite time.
- It should produce at least one output.
- It should take zero or more input.
- It should be deterministic means giving the same output for the same input case.
- Every step in the algorithm must be effective i.e. every step should do some work.

Types of Algorithms:

There are several types of algorithms available. Some important algorithms are:

1. Brute Force Algorithm: It is the simplest approach for a problem. A brute force algorithm is the first approach that comes to finding when we see a problem.

2. Recursive Algorithm: A recursive algorithm is based on recursion. In this case, a problem is broken into several sub-parts and called the same function again and again.

3. Backtracking Algorithm: The backtracking algorithm basically builds the solution by searching among all possible solutions. Using this algorithm, we keep on building the solution following criteria. Whenever a solution fails we trace back to

the failure point and build on the next solution and continue this process till we find the solution or all possible solutions are looked after.

4. Searching Algorithm: Searching algorithms are the ones that are used for searching elements or groups of elements from a particular data structure. They can be of different types based on their approach or the data structure in which the element should be found.

5. Sorting Algorithm: Sorting is arranging a group of data in a particular manner according to the requirement. The algorithms which help in performing this function are called sorting algorithms. Generally sorting algorithms are used to sort groups of data in an increasing or decreasing manner.

6. Hashing Algorithm: Hashing algorithms work similarly to the searching algorithm. But they contain an index with a key ID. In hashing, a key is assigned to specific data.

7. Divide and Conquer Algorithm: This algorithm breaks a problem into sub-problems, solves a single sub-problem and merges the solutions together to get the final solution. It consists of the following three steps:

- Divide
- Solve
- Combine

8. Greedy Algorithm: In this type of algorithm the solution is built part by part. The solution of the next part is built based on the immediate benefit of the next part. The one solution giving the most benefit will be chosen as the solution for the next part.

9. Dynamic Programming Algorithm: This algorithm uses the concept of using the already found solution to avoid repetitive calculation of the same part of the problem. It divides the problem into smaller overlapping subproblems and solves them.

10. Randomized Algorithm: In the randomized algorithm we use a random number so it gives immediate benefit. The random number helps in deciding the expected outcome.

Advantages of Algorithms:

- It is easy to understand.
- An algorithm is a step-wise representation of a solution to a given problem.
- In Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

Disadvantages of Algorithms:

- Writing an algorithm takes a long time so it is time-consuming.
- Understanding complex logic through algorithms can be very difficult.
- Branching and Looping statements are difficult to show in Algorithms

How to Design an Algorithm?

In order to write an algorithm, the following things are needed as a pre-requisite:

1. The problem that is to be solved by this algorithm i.e. clear problem definition.
2. The constraints of the problem must be considered while solving the problem.
3. The input to be taken to solve the problem.
4. The output to be expected when the problem is solved.
5. The solution to this problem, is within the given constraints.

Then the algorithm is written with the help of the above parameters such that it solves the problem.
Example: Consider the example to add three numbers and print the sum.

- Step 1: Fulfilling the pre-requisites
  As discussed above, in order to write an algorithm, its pre-requisites must be fulfilled.
  1. The problem that is to be solved by this algorithm: Add 3 numbers and print their sum.
  2. The constraints of the problem that must be considered while solving the problem: The numbers must contain only digits and no other characters.
  3. The input to be taken to solve the problem: The three numbers to be added.
  4. The output to be expected when the problem is solved: The sum of the three numbers taken as the input i.e. a single integer value.
  5. The solution to this problem, in the given constraints: The solution consists of adding the 3 numbers. It can be done with the help of '+' operator, or bit-wise, or any other method.

- Step 2: Designing the algorithm
  Now let's design the algorithm with the help of the above pre-requisites:
  Algorithm to add 3 numbers and print their sum:

  1. START
  2. Declare 3 integer variables num1, num2 and num3.
  3. Take the three numbers, to be added, as inputs in variables num1, num2, and num3 respectively.
  4. Declare an integer variable sum to store the resultant sum of the 3 numbers.
  5. Add the 3 numbers and store the result in the variable sum.
  6. Print the value of the variable sum
  7. END

- Step 3: Testing the algorithm by implementing it.
  In order to test the algorithm, let's implement it in C language.

Eg;

```
// C++ program to add three numbers
// with the help of above designed
// algorithm
#include <bits/stdc++.h>
using namespace std;

int main()
{

    // Variables to take the input of
    // the 3 numbers
    int num1, num2, num3;

    // Variable to store the resultant sum
    int sum;
```

```
        // Take the 3 numbers as input
        cout << "Enter the 1st number: ";
        cin >> num1;
        cout << " " << num1 << endl;

        cout << "Enter the 2nd number: ";
        cin >> num2;
        cout << " " << num2 << endl;

        cout << "Enter the 3rd number: ";
        cin >> num3;
        cout << " " << num3;

        // Calculate the sum using + operator
        // and store it in variable sum
        sum = num1 + num2 + num3;

        // Print the sum
        cout << "\nSum of the 3 numbers is: "
                << sum;

        return 0;
    }
```

OUTPUT:

```
Enter the 1st number: 0
Enter the 2nd number: 0
Enter the 3rd number: -1577141152

Sum of the 3 numbers is: -1577141152
```

One problem, many solutions: The solution to an algorithm can be or cannot be more than one. It means that while implementing the algorithm, there can be more than one method to implement it. For example, in the above problem to add 3 numbers, the sum can be calculated in many ways like:

- + operator
- Bit-wise operators
- . . etc

How to analyse an Algorithm?

For a standard algorithm to be good, it must be efficient. Hence the efficiency of an algorithm must be checked and maintained. It can be in two stages:

1. Priori Analysis: “Priori” means “before”. Hence Priori analysis means checking the algorithm before its implementation. In this, the algorithm is checked when it is written in the form of theoretical steps. This Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation. This is done usually by the algorithm designer. This analysis is independent of the type of hardware and language of the compiler. It gives the approximate answers for the complexity of the program.
2. Posterior Analysis: “Posterior” means “after”. Hence Posterior analysis means checking the algorithm after its implementation. In this, the algorithm is checked by implementing it in any programming language and executing it. This analysis helps to get the actual and real analysis report about correctness(for every possible input/s if it shows/returns correct output or not), space required, time consumed etc. That is, it is dependent on the language of the compiler and the type of hardware used.

What is Algorithm complexity and how to find it?

An algorithm is defined as complex based on the amount of Space and Time it consumes. Hence the Complexity of an algorithm refers to the measure of the Time that it will need to execute and get the expected output, and the Space it will need to store all the data (input, temporary data and output). Hence these two factors define the efficiency of an algorithm.
The two factors of Algorithm Complexity are:

- Time Factor: Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- Space Factor: Space is measured by counting the maximum memory space required by the algorithm to run/execute.

Therefore the complexity of an algorithm can be divided into two types:

1. Space Complexity: The space complexity of an algorithm refers to the amount of memory required by the algorithm to store the variables and get the result. This can be for inputs, temporary operations, or outputs.

How to calculate Space Complexity?
The space complexity of an algorithm is calculated by determining the following 2 components:

- Fixed Part: This refers to the space that is definitely required by the algorithm. For example, input variables, output variables, program size, etc.
- Variable Part: This refers to the space that can be different based on the implementation of the algorithm. For example, temporary variables, dynamic memory allocation, recursion stack space, etc.
  Therefore Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I.

Example: Consider the below algorithm for Linear Search

Step 1: START
Step 2: Get n elements of the array in arr and the number to be searched in x
Step 3: Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
Step 4: If x matches with an element, Print True.
Step 5: If x doesn't match with any of the elements, Print False.
Step 6: END
Here, There are 2 variables arr[], and x, where the arr[] is the variable part of n elements and x is the fixed part. Hence S(P) = 1+n. So, the space complexity depends on n(number of elements). Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

2. Time Complexity: The time complexity of an algorithm refers to the amount of time that is required by the algorithm to execute and get the result. This can be for normal operations, conditional if-else statements, loop statements, etc.

How to calculate Time Complexity?
The time complexity of an algorithm is also calculated by determining the following 2 components:

- Constant time part: Any instruction that is executed just once comes in this part. For example, input, output, if-else, switch, arithmetic operations etc.
- Variable Time Part: Any instruction that is executed more than once, say n times, comes in this part. For example, loops, recursion, etc.

  Therefore Time complexity of any algorithm P is T(P) = C + TP(I), where C is the constant time part and TP(I) is the variable part of the algorithm, which depends on the instance characteristic I.

Example: In the algorithm of Linear Search above, the time complexity is calculated as follows:

Step 1: –Constant Time
Step 2: — Variable Time (Taking n inputs)
Step 3: –Variable Time (Till the length of the Array (n) or the index of the found element)
Step 4: –Constant Time
Step 5: –Constant Time
Step 6: –Constant Time
Hence, T(P) = 5 + n, which can be said as T(n).

How to express an Algorithm?

1. Natural Language :- Here we express the Algorithm in natural English language. It is too hard to understand the algorithm from it.
2. Flow Chat :- Here we express the Algorithm by making graphical/pictorial representation of it. It is easier to understand than Natural Language.
3. Pseudo Code :- Here we express the Algorithm in the form of annotations and informative text written in plain English which is very much similar to the real code but as it has no syntax like any of the programming language, it can't be compiled or interpreted by the computer. It is the best way to express an algorithm because it can be understood by even a layman with some school level programming knowledge.

How Discrete Mathematics and Data structures Related?

Every single data structure you intend to build must be rigorously defined. Each method on a data structure must be proven rigorously to be correct. The time and space complexity of each method must be rigorously proven to follow a particular asymptotic bound. This all involves some nontrivial discrete mathematics.

How is discrete math used in algorithms?

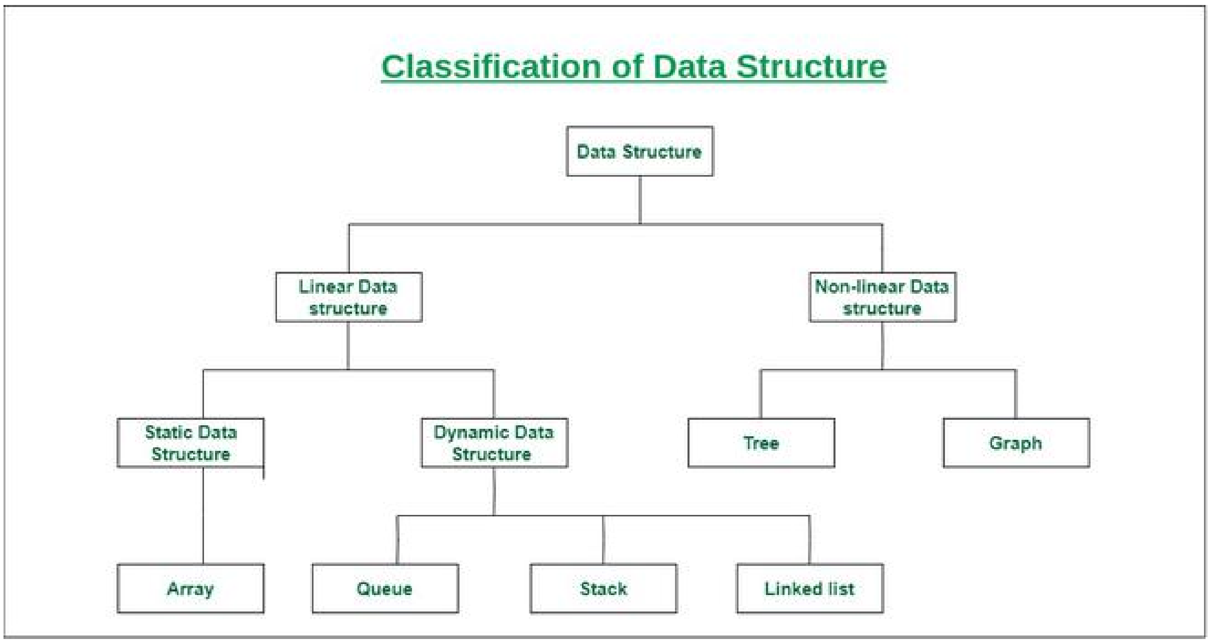
Examples of common discrete mathematics algorithms include: Searching Algorithms to search for an item in a data set or data structure like a tree. Sorting Algorithms to sort items in a specific order. Insertion and Deletion Algorithms to insert or delete item in a data structure such as a tree or list.

What is Data Structure:

A data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

A data structure is not only used for organizing the data. It is also used for processing, retrieving, and storing data. There are different basic and advanced types of data structures that are used in almost every program or software system that has been developed. So we must have good knowledge about data structures.

Classification of Data Structure:

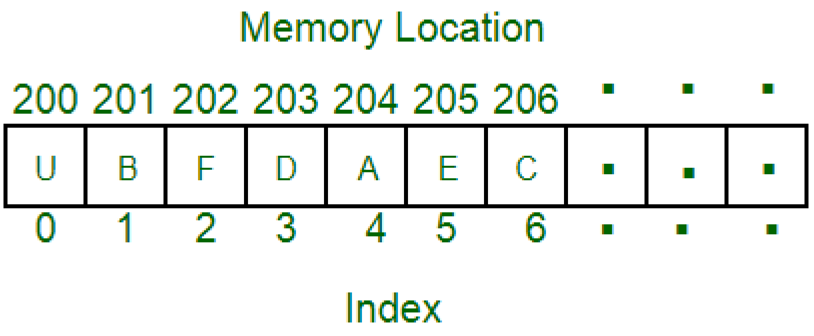


Classification of Data Structure

- Linear data structure: Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure.
  Examples of linear data structures are array, stack, queue, linked list, etc.
    - Static data structure: Static data structure has a fixed memory size. It is easier to access the elements in a static data structure.
      An example of this data structure is an array.
    - Dynamic data structure: In dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code.
      Examples of this data structure are queue, stack, etc.

- Non-linear data structure: Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can't traverse all the elements in a single run only.
  Examples of non-linear data structures are trees and graphs.

For example, we can store a list of items having the same data-type using the *array* data structure.



Array Data Structure

What are path finding algorithms?

Path finding algorithms build on top of graph search algorithms and explore routes between nodes, starting at one node and traversing through relationships until the destination has been reached. These algorithms find the cheapest path in terms of the number of hops or weight. Weights can be anything measured, such as time, distance, capacity, or cost

.

Shortest Path

The Shortest Path algorithm calculates the shortest (weighted) path between a pair of nodes. Shortest path is considered to be one of the classical graph problems and has been researched as far back as the 19th century. It has the following use cases:

- Finding directions between physical locations. This is the most common usage, and web mapping tools such as Google Maps use the shortest path algorithm, or a variant of it, to provide driving directions.
- Social networks can use the algorithm to find the degrees of separation between people. For example, when you view someone's profile on LinkedIn, it will indicate how many people separate you in the connections graph, as well as listing your mutual connections.
-

A* Algorithm-

The A* (pronounced "A-Star") Shortest Path algorithm computes the shortest path between two nodes. A* is an informed search algorithm as it uses a heuristic function to guide the graph traversal. The algorithm supports weighted graphs with positive relationship weights.

Unlike Dijkstra's shortest path algorithm, the next node to search from is not solely picked on the already computed distance. Instead, the algorithm combines the already computed distance with the result of a heuristic function. That function takes a node as input and returns a value that corresponds to the cost to reach the target node from that node. In each iteration, the graph traversal is continued from the node with the lowest combined cost.

A* Algorithm is one of the best and popular techniques used for path finding and graph traversals.

A lot of games and web-based maps use this algorithm for finding the shortest path efficiently.

It is essentially a best first search algorithm.

Working-

A* Algorithm works as-

It maintains a tree of paths originating at the start node.

It extends those paths one edge at a time.

It continues until its termination criterion is satisfied.

A* Algorithm extends the path that minimizes the following function-

$$f(n) = g(n) + h(n)$$

Here,

‘n’ is the last node on the path

g(n) is the cost of the path from start node to node ‘n’

h(n) is a heuristic function that estimates cost of the cheapest path from node ‘n’ to the goal node

Why A* Search Algorithm?

A* Search Algorithm is a simple and efficient search algorithm that can be used to find the optimal path between two nodes in a graph. It will be used for the shortest path finding. It is an extension of Dijkstra’s shortest path algorithm (Dijkstra’s Algorithm). The extension here is that, instead of using a priority queue to store all the elements, we use heaps (binary trees) to store them. The A* Search Algorithm also uses a heuristic function that provides additional information regarding how far away from the goal node we are. This function is used in conjunction with the f-heap data structure in order to make searching more efficient.

Explanation

In the event that we have a grid with many obstacles and we want to get somewhere as rapidly as possible, the A* Search Algorithms are our savior. From a given starting cell, we can get to the target cell as quickly as possible. It is the sum of two variables' values that determines the node it picks at any point in time.

At each step, it picks the node with the smallest value of 'f' (the sum of 'g' and 'h') and processes that node/cell. 'g' and 'h' is defined as simply as possible below:

- 'g' is the distance it takes to get to a certain square on the grid from the starting point, following the path we generated to get there.
- 'h' is the heuristic, which is the estimation of the distance it takes to get to the finish line from that square on the grid.

Heuristics are basically educated guesses. It is crucial to understand that we do not know the distance to the finish point until we find the route since there are so many things that might get in the way (e.g., walls, water, etc.). In the coming sections, we will dive deeper into how to calculate the heuristics.

Let us now look at the detailed algorithm of A*.

Algorithm

Initial condition - we create two lists - Open List and Closed List.

Now, the following steps need to be implemented -

- The open list must be initialized.
- Put the starting node on the open list (leave its f at zero). Initialize the closed list.
- Follow the steps until the open list is non-empty:

1. Find the node with the least f on the open list and name it "q".
2. Remove Q from the open list.
3. Produce q's eight descendants and set q as their parent.
4. For every descendant:

i) If finding a successor is the goal, cease looking

ii)Else, calculate g and h for the successor.

successor.g = q.g + the calculated distance between the successor and the q.

successor.h = the calculated distance between the successor and the goal. We will cover three heuristics to do this: the Diagonal, the Euclidean, and the Manhattan heuristics.

successor.f = successor.g plus successor.h

iii) Skip this successor if a node in the OPEN list with the same location as it but a lower f value than the successor is present.

iv) Skip the successor if there is a node in the CLOSED list with the same position as the successor but a lower f value; otherwise, add the node to the open list end (for loop).

- Push Q into the closed list and end the while loop.

We will now discuss how to calculate the Heuristics for the nodes.

Heuristics

We can easily calculate g, but how do we actually calculate h?

There are two methods that we can use to calculate the value of h:

1. Determine h's exact value (which is certainly time-consuming).

(or)

2. Utilize various techniques to approximate the value of h. (less time-consuming).

Let us discuss both methods.

Exact Heuristics

Although we can obtain exact values of h, doing so usually takes a very long time.

The ways to determine h's precise value are listed below.

1. Before using the A* Search Algorithm, pre-calculate the distance between every pair of cells.

2. Using the distance formula/Euclidean Distance, we may directly determine the precise value of h in the absence of blocked cells or obstructions.

Let us look at how to calculate Approximation Heuristics.

Approximation Heuristics

To determine h, there are typically three approximation heuristics:

1. Manhattan Distance

The Manhattan Distance is the total of the absolute values of the discrepancies between the x and y coordinates of the current and the goal cells.

The formula is summarized below -

h = abs (curr_cell.x – goal.x) +

    abs (curr_cell.y – goal.y)

We must use this heuristic method when we are only permitted to move in four directions - top, left, right, and bottom.

Let us now take a look at the Diagonal Distance method to calculate the heuristic.

2. Diagonal Distance

It is nothing more than the greatest absolute value of differences between the x and y coordinates of the current cell and the goal cell.

This is summarized below in the following formula -

dx = abs(curr_cell.x – goal.x)

dy = abs(curr_cell.y – goal.y)

h = D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)

where D is the length of every node (default = 1) and D2 is the diagonal

We use this heuristic method when we are permitted to move only in eight directions, like the King's moves in Chess.

Let us now take a look at the Euclidean Distance method to calculate the heuristic.

3. Euclidean Distance

The Euclidean Distance is the distance between the goal cell and the current cell using the distance formula:

h = sqrt ((curr_cell.x – goal.x)^2 +

        (curr_cell.y – goal.y)^2)

We use this heuristic method when we are permitted to move in any direction of our choice.

The Basic Concept of A* Algorithm

A heuristic algorithm sacrifices optimality, with precision and accuracy for speed, to solve problems faster and more efficiently.

All graphs have different nodes or points which the algorithm has to take, to reach the final node. The paths between these nodes all have a numerical value, which is considered as the weight of the path. The total of all paths transverse gives you the cost of that route.

Initially, the Algorithm calculates the cost to all its immediate neighboring nodes,n, and chooses the one incurring the least cost. This process repeats until no new nodes can be chosen and all paths have been traversed. Then, you should consider the best path among them. If f(n) represents the final cost, then it can be denoted as :

f(n) = g(n) + h(n), where :

g(n) = cost of traversing from one node to another. This will vary from node to node

h(n) = heuristic approximation of the node's value. This is not a real value but an approximation cost

How Does the A* Algorithm Work?

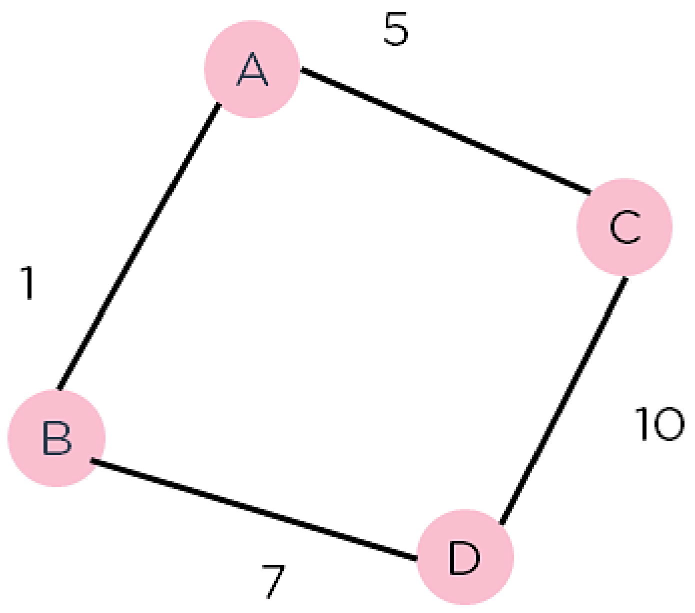


Figure 2: Weighted Graph 2

Consider the weighted graph depicted above, which contains nodes and the distance between them. Let's say you start from A and have to go to D.

Now, since the start is at the source A, which will have some initial heuristic value. Hence, the results are

f(A) = g(A) + h(A)
f(A) = 0 + 6 = 6

Next, take the path to other neighbouring vertices :

f(A-B) = 1 + 4

f(A-C) = 5 + 2

Now take the path to the destination from these nodes, and calculate the weights :

f(A-B-D) = (1+ 7) + 0

f(A-C-D) = (5 + 10) + 0

It is clear that node B gives you the best path, so that is the node you need to take to reach the destination.

THANKS!