8.Concept of Heuristic Search in AI

**Heuristic Search:**

A Heuristic is a technique to solve a problem faster than classic methods, or to find an approximate solution when classic methods cannot. This is a kind of a shortcut as we often trade one of optimality, completeness, accuracy, or precision for speed.

A Heuristic (or a heuristic function) takes a look at search algorithms. At each branching step, it evaluates the available information and makes a decision on which branch to follow.

It does so by ranking alternatives. The Heuristic is any device that is often effective but will not guarantee work in every case.

So why do we need heuristics? One reason is to produce, in a reasonable amount of time, a solution that is good enough for the problem in question. It doesn’t have to be the best- an approximate solution will do since this is fast enough.

**Heuristic Search Techniques in AI**

Other names for these are Informed Search, Heuristic Search, and Heuristic Control Strategy. These are effective if applied correctly to the right types of tasks and usually demand domain-specific information.

We need this extra information to compute preference among child nodes to explore and expand. Each node has a heuristic function associated with it. Examples are Best First Search (BFS) and A\*.

Before we move on to describe certain techniques, let’s first take a look at the ones we generally observe. Below, we name a few.

* Best-First Search
* A\* Search
* Bidirectional Search
* Tabu Search
* Beam Search
* Simulated Annealing
* Hill Climbing
* Constraint Satisfaction Problems

## Constraint Satisfaction Problems (CSP)

Let’s talk of a magic square. This is a sequence of numbers- usually integers- arranged in a square grid. The numbers in each row, each column, and each diagonal all add up to a constant which we call the *Magic Constant*. Let’s implement this with Python.

def magic\_square(matrix):

size=len(matrix[0])

sum\_list=[]

for col in range(size): #Vertical sum

sum\_list.append(sum(row[col] for row in matrix))

sum\_list.extend([sum(lines) for lines in matrix])#Horizontal sum

result1=0

for i in range(0,size):

result1+=matrix[i][i]

sum\_list.append(result1)

result2=0

for i in range(size-1,-1,-1):

result2+=matrix[i][i]

sum\_list.append(result2)

if len(set(sum\_list))>1:

return False

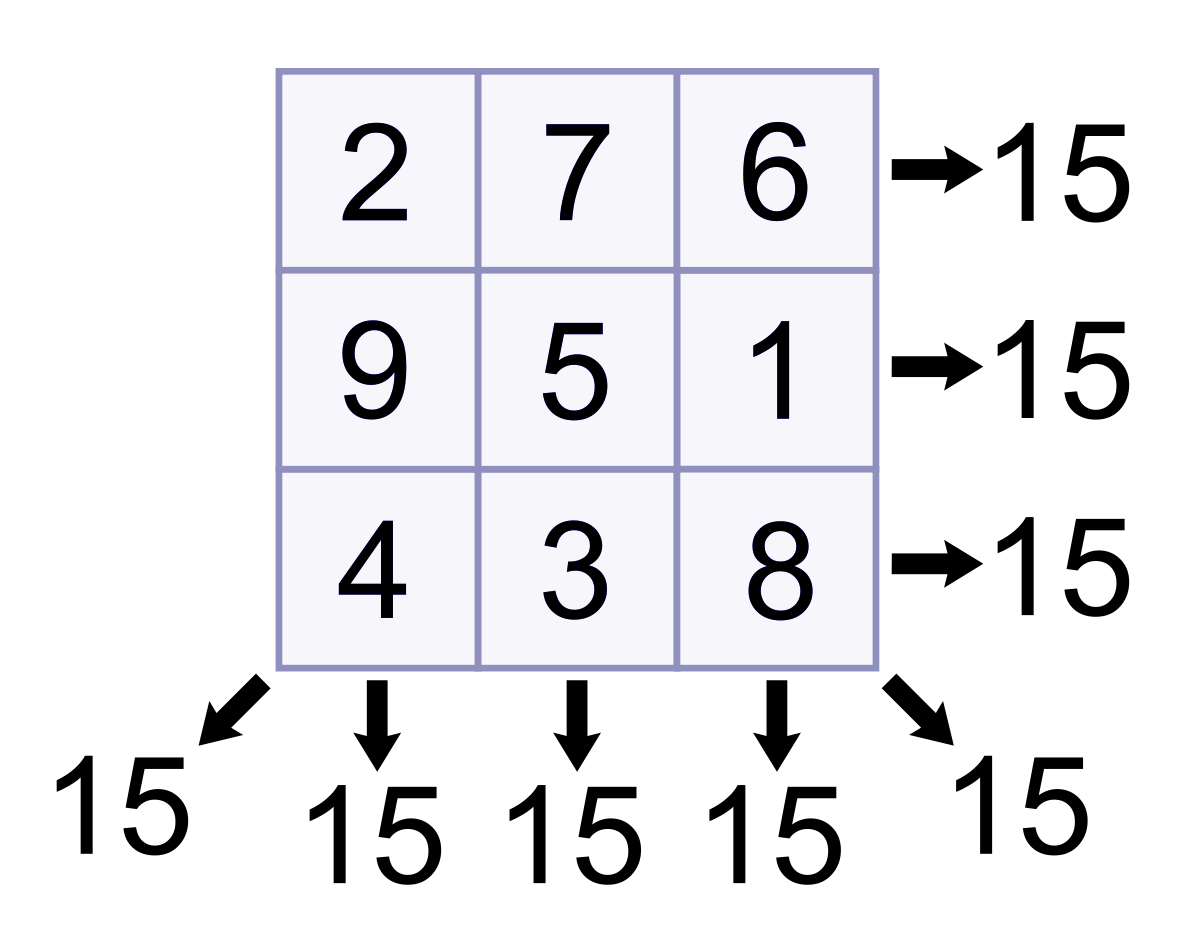
return True

>>> magic\_square([[1,2,3],[4,5,6],[7,8,9]])

False

>>> magic\_square([[2,7,6],[9,5,1],[4,3,8]])

True



**A\* Search Algorithm and Its Basic Concepts**

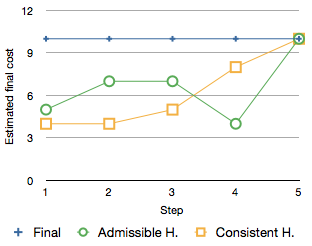
A\* algorithm works based on heuristic methods, and this helps achieve optimality. A\* is a different form of the best-first algorithm. Optimality empowers an algorithm to find the best possible solution to a problem. Such algorithms also offer completeness; if there is any solution possible to an existing problem, the algorithm will definitely find it.

When A\* enters into a problem, firstly, it calculates the cost to travel to the neighboring nodes and chooses the node with the lowest cost. If The f(n) denotes the cost, A\* chooses the node with the lowest f(n) value. Here ‘n’ denotes the neighboring nodes. The calculation of the value can be done as shown below:

f(n)=g(n)+h(n)f(n)=g(n)+h(n)  
g(n) = shows the shortest path’s value from the starting node to node n  
h(n) = The heuristic approximation of the value of the node

The heuristic value has an important role in the efficiency of the A\* algorithm. To find the best solution, you might have to use different heuristic functions according to the type of the problem. However, the creation of these functions is a difficult task, and this is the basic problem we face in AI.

**What is a Heuristic Function?**



A heuristic is simply called a heuristic function that helps rank the alternatives given in a search algorithm at each of its steps. It can either produce a result on its own or work in conjugation with a given algorithm to create a result. Essentially, a heuristic function helps algorithms to make the best decision faster and more efficiently. This ranking is based on the best available information and helps the algorithm decide the best possible branch to follow. Admissibility and consistency are the two fundamental properties of a heuristic function.

**Admissibility of the Heuristic Function**

A heuristic function is admissible if it can effectively estimate the real distance between a node ‘n’ and the end node. It never overestimates; if it ever does, it will be denoted by ‘d’, which also denotes the accuracy of the solution.

**Consistency of the Heuristic Function**

A heuristic function is consistent if the estimate of a given heuristic function turns out to be equal to or less than the distance between the goal (n) and a neighbor and the cost calculated to reach that neighbor.

A\* is indeed a very powerful algorithm used to increase the performance of artificial intelligence. It is one of the most popular search algorithms in AI. The sky is the limit when it comes to the potential of this algorithm. However, the efficiency of an A\* algorithm highly depends on the quality of its heuristic function. Wonder why this algorithm is preferred and used in many software systems? There is no single facet of AI where the A\*algorithm has not found its application. From search optimization to games, robotics, and machine learning, the A\* algorithm is an inevitable part of a smart program.

**Implementation with Python**

In this section, we are going to find out how the A\* search algorithm can be used to find the most cost-effective path in a graph. Consider the following graph below.

Implementation with Python


The numbers written on edges represent the distance between the nodes, while the numbers written on nodes represent the heuristic values. Let us find the most cost-effective path to reach from start state A to final state G using the A\* Algorithm.

Let’s start with node A. Since A is a starting node, therefore, the value of g(x) for A is zero, and from the graph, we get the heuristic value of A is 11, therefore

g(x) + h(x) = f(x)

0+ 11 =11

Thus for A, we can write

A=11

Now from A, we can go to point B or point E, so we compute f(x) for each of them

A → B = 2 + 6 = 8

A → E = 3 + 6 = 9

Since the cost for A → B is less, we move forward with this path and compute the f(x) for the children nodes of B

Since there is no path between C and G, the heuristic cost is set to infinity or a very high value

A → B → C = (2 + 1) + 99= 102

A → B → G = (2 + 9 ) + 0 = 11

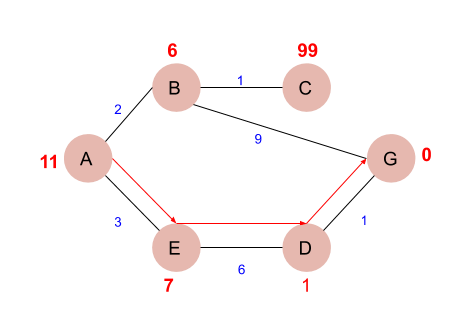
Here the path A → B → G has the least cost but it is still more than the cost of A → E, thus we explore this path further

A → E → D = (3 + 6) + 1 = 10

Comparing the cost of A → E → D with all the paths we got so far and as this cost is least of all we move forward with this path. And compute the f(x) for the children of D

A → E → D → G = (3 + 6 + 1) +0 =10

Now comparing all the paths that lead us to the goal, we conclude that A → E → D → G is the most cost-effective path to get from A to G.



Next, we write a program in Python that can find the most cost-effective path by using the a-star algorithm.

First, we create two sets, viz- open and close. The open contains the nodes that have been visited, but their neighbors are yet to be explored. On the other hand, close contains nodes that, along with their neighbors, have been visited.

def aStarAlgo(start\_node, stop\_node):

        open\_set = set(start\_node)

        closed\_set = set()

        g = {} #store distance from starting node

        parents = {}# parents contains an adjacency map of all nodes

        #ditance of starting node from itself is zero

        g[start\_node] = 0

        #start\_node is root node i.e it has no parent nodes

        #so start\_node is set to its own parent node

        parents[start\_node] = start\_node

        while len(open\_set) > 0:

            n = None

            #node with lowest f() is found

            for v in open\_set:

                if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):

                    n = v

            if n == stop\_node or Graph\_nodes[n] == None:

                pass

            else:

                for (m, weight) in get\_neighbors(n):

                    #nodes 'm' not in first and last set are added to first

                    #n is set its parent

                    if m not in open\_set and m not in closed\_set:

                        open\_set.add(m)

                        parents[m] = n

                        g[m] = g[n] + weight

                    #for each node m,compare its distance from start i.e g(m) to the

                    #from start through n node

                    else:

                        if g[m] > g[n] + weight:

                            #update g(m)

                            g[m] = g[n] + weight

                            #change parent of m to n

                            parents[m] = n

                            #if m in closed set,remove and add to open

                            if m in closed\_set:

                                closed\_set.remove(m)

                                open\_set.add(m)

            if n == None:

                print('Path does not exist!')

                return None

            # if the current node is the stop\_node

            # then we begin reconstructin the path from it to the start\_node

            if n == stop\_node:

                path = []

                while parents[n] != n:

                    path.append(n)

                    n = parents[n]

                path.append(start\_node)

                path.reverse()

                print('Path found: {}'.format(path))

                return path

            # remove n from the open\_list, and add it to closed\_list

            # because all of his neighbors were inspected

            open\_set.remove(n)

            closed\_set.add(n)

        print('Path does not exist!')

        return None

#define fuction to return neighbor and its distance

#from the passed node

def get\_neighbors(v):

    if v in Graph\_nodes:

        return Graph\_nodes[v]

    else:

        return None

#for simplicity we ll consider heuristic distances given

#and this function returns heuristic distance for all nodes

def heuristic(n):

        H\_dist = {

            'A': 11,

            'B': 6,

            'C': 99,

            'D': 1,

            'E': 7,

            'G': 0,

        }

        return H\_dist[n]

#Describe your graph here

Graph\_nodes = {

    'A': [('B', 2), ('E', 3)],

    'B': [('C', 1),('G', 9)],

    'C': None,

    'E': [('D', 6)],

    'D': [('G', 1)],

}

aStarAlgo('A', 'G')

Output:

Path Found: [ 'A','E','D','G']