**LAB ASSIGNMENT 02**

**COMSATS University Islamabad**

Sahiwal Campus



***Usama Sarwar***

FA17-BS(CS)-090-B

***Mr. Waqar***

Artificial Intelligence

November 23, 2020

Table of Contents

[1. Short Questions 1](#_Toc57045647)

[1.1 Informed and Uninformed Searches 1](#_Toc57045648)

[1.1.1 Informed Search 1](#_Toc57045649)

[1.1.2 Uninformed Search 1](#_Toc57045650)

[1.2 Uniform-Cost Search (Dijkstra for large Graphs) 2](#_Toc57045651)

[1.3 Bidirectional Search Algorithm 2](#_Toc57045652)

[1.4 Difference between graph and tree traversal 2](#_Toc57045653)

[1.5 Heuristic Search 2](#_Toc57045654)

[1.6 Adversarial Search 3](#_Toc57045655)

[2. Codes 3](#_Toc57045656)

[2.1 Depth-Limited Search 3](#_Toc57045657)

[2.1.1 Console Output 4](#_Toc57045658)

[2.2 Iterative deepening search 4](#_Toc57045659)

[2.2.1 Console Output 6](#_Toc57045660)

[2.3 A-Star Search Algorithm 6](#_Toc57045661)

[2.3.1 Console Output 10](#_Toc57045662)

[2.4 Genetic Algorithm 10](#_Toc57045663)

[2.4.1 Console Output 13](#_Toc57045664)

[2.5 Min-Max and Alpha-Beta pruning 14](#_Toc57045665)

[2.5.1 Console Output 15](#_Toc57045666)

[2.6 Hill climbing algorithm 15](#_Toc57045667)

[2.6.1 Console Output 16](#_Toc57045668)

[2.6.2 Screenshoots 18](#_Toc57045669)

# Short Questions

## Informed and Uninformed Searches

### Informed Search

Informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach to goal node, etc. This knowledge help agents to explore less to the search space and find more efficiently the goal node. The informed search algorithm is more useful for large search space. Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.

**Heuristics function**: Heuristic is a function which is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal. The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time. Heuristic function estimates how close a state is to the goal. It is represented by h(n), and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.

#### Types

1. Pure Heuristic Search
2. Best-first Search Algorithm (Greedy Search)
3. A\* Search Algorithm

### Uninformed Search

Uninformed search is a class of general-purpose search algorithms which operates in brute force-way. Uninformed search algorithms do not have additional information about state or search space other than how to traverse the tree, so it is also called blind search.

#### Types

1. Breadth-first Search
2. Depth-first Search
3. Depth-limited Search
4. Iterative deepening depth-first search
5. Uniform cost search
6. Bidirectional Search

## Uniform-Cost Search (Dijkstra for large Graphs)

Uniform-Cost Search is a variant of Dijikstra’s algorithm. Here, instead of inserting all vertices into a priority queue, we insert only source, then one by one insert when needed. In every step, we check if the item is already in priority queue (using visited array). If yes, we perform decrease key, else we insert it. This variant of Dijsktra is useful for infinite graphs and those graph which are too large to represent in the memory. Uniform-Cost Search is mainly used in Artificial Intelligence.

## Bidirectional Search Algorithm

Bidirectional search is a graph search algorithm that finds a shortest path from an initial vertex to a goal vertex in a directed graph. It runs two simultaneous searches: one forward from the initial state, and one backward from the goal, stopping when the two meet.

## Difference between graph and tree traversal

1. In a tree there exist only one path between any two vertices whereas a graph can have unidirectional and bidirectional paths between the nodes.
2. In the tree, there is exactly one root node, and every child can have only one parent. As against, in a graph, there is no concept of the root node.
3. A tree cannot have loops and self-loops while graph can have loops and self-loops.
4. Graphs are more complicated as it can have loops and self-loops. In contrast, trees are simple as compared to the graph.
5. The tree is traversed using pre-order, in-order and post-order techniques. On the other hand, for graph traversal, we use BFS (Breadth First Search) and DFS (Depth First Search).
6. A tree can have n-1 edges. On the contrary, in the graph, there is no predefined number of edges, and it depends on the graph.
7. A tree has a hierarchical structure whereas graph has a network model.

## Heuristic Search

A heuristic function, also called simply a heuristic, is a function that ranks alternatives in search algorithms at each branching step based on available information to decide which branch to follow. For example, it may approximate the exact solution.

**Heuristics function:** Heuristic is a function which is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal. The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time. Heuristic function estimates how close a state is to the goal. It is represented by h(n), and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.

Admissibility of the heuristic function is given as:

h(n) <= h\*(n)

Here h(n) is heuristic cost, and h\*(n) is the estimated cost. Hence heuristic cost should be less than or equal to the estimated cost.

## Adversarial Search

Adversarial search is search when there is an "enemy" or "opponent" changing the state of the problem every step in a direction you do not want. Examples: Chess, business, trading, war. You change state, but then you don't control the next state. Opponent will change the next state in a way: unpredictable.

# Codes

## Depth-Limited Search

# Python program to print DFS traversal from a given

# given graph

from collections import defaultdict

# This class represents a directed graph using adjacency

# list representation

class Graph:

    def \_\_init\_\_(self,vertices):

        # No. of vertices

        self.V = vertices

        # default dictionary to store graph

        self.graph = defaultdict(list)

    # function to add an edge to graph

    def addEdge(self,u,v):

        self.graph[u].append(v)

    # A function to perform a Depth-Limited search

    # from given source 'src'

    def DLS(self,src,target,maxDepth):

        if src == target : return True

        # If reached the maximum depth, stop recursing.

        if maxDepth <= 0 : return False

        # Recur for all the vertices adjacent to this vertex

        for i in self.graph[src]:

                if(self.DLS(i,target,maxDepth-1)):

                    return True

        return False

    # IDDFS to search if target is reachable from v.

    # It uses recursive DLS()

    def IDDFS(self,src, target, maxDepth):

        # Repeatedly depth-limit search till the

        # maximum depth

        for i in range(maxDepth):

            if (self.DLS(src, target, i)):

                return True

        return False

# Create a graph given in the above diagram

g = Graph (7);

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 3)

g.addEdge(1, 4)

g.addEdge(2, 5)

g.addEdge(2, 6)

target = 6; maxDepth = 3; src = 0

if g.IDDFS(src, target, maxDepth) == True:

    print ("Target is reachable from source " +

        "within max depth")

else :

    print ("Target is NOT reachable from source " +

        "within max depth")

### Console Output

Target is reachable from source within max depth

## Iterative deepening search

// C++ program to search if a target node is reachable from

// a source with given max depth.

#include<bits/stdc++.h>

using namespace std;

// Graph class represents a directed graph using adjacency

// list representation.

class Graph

{

    int V;    // No. of vertices

    // Pointer to an array containing

    // adjacency lists

    list<int> \*adj;

    // A function used by IDDFS

    bool DLS(int v, int target, int limit);

public:

    Graph(int V);   // Constructor

    void addEdge(int v, int w);

    // IDDFS traversal of the vertices reachable from v

    bool IDDFS(int v, int target, int max\_depth);

};

Graph::Graph(int V)

{

    this->V = V;

    adj = new list<int>[V];

}

void Graph::addEdge(int v, int w)

{

    adj[v].push\_back(w); // Add w to v’s list.

}

// A function to perform a Depth-Limited search

// from given source 'src'

bool Graph::DLS(int src, int target, int limit)

{

    if (src == target)

        return true;

    // If reached the maximum depth, stop recursing.

    if (limit <= 0)

        return false;

    // Recur for all the vertices adjacent to source vertex

    for (auto i = adj[src].begin(); i != adj[src].end(); ++i)

       if (DLS(\*i, target, limit-1) == true)

          return true;

     return false;

}

// IDDFS to search if target is reachable from v.

// It uses recursive DFSUtil().

bool Graph::IDDFS(int src, int target, int max\_depth)

{

    // Repeatedly depth-limit search till the

    // maximum depth.

    for (int i = 0; i <= max\_depth; i++)

       if (DLS(src, target, i) == true)

          return true;

    return false;

}

// Driver code

int main()

{

    // Let us create a Directed graph with 7 nodes

    Graph g(7);

    g.addEdge(0, 1);

    g.addEdge(0, 2);

    g.addEdge(1, 3);

    g.addEdge(1, 4);

    g.addEdge(2, 5);

    g.addEdge(2, 6);

    int target = 6, maxDepth = 3, src = 0;

    if (g.IDDFS(src, target, maxDepth) == true)

        cout << "Target is reachable from source "

                "within max depth";

    else

        cout << "Target is NOT reachable from source "

                "within max depth";

    return 0;

}

### Console Output

Target is reachable from source within max depth

## A-Star Search Algorithm

# This class represent a graph

class Graph:

    # Initialize the class

    def \_\_init\_\_(self, graph\_dict=None, directed=True):

        self.graph\_dict = graph\_dict or {}

        self.directed = directed

        if not directed:

            self.make\_undirected()

    # Create an undirected graph by adding symmetric edges

    def make\_undirected(self):

        for a in list(self.graph\_dict.keys()):

            for (b, dist) in self.graph\_dict[a].items():

                self.graph\_dict.setdefault(b, {})[a] = dist

    # Add a link from A and B of given distance, and also add the inverse link if the graph is undirected

    def connect(self, A, B, distance=1):

        self.graph\_dict.setdefault(A, {})[B] = distance

        if not self.directed:

            self.graph\_dict.setdefault(B, {})[A] = distance

    # Get neighbors or a neighbor

    def get(self, a, b=None):

        links = self.graph\_dict.setdefault(a, {})

        if b is None:

            return links

        else:

            return links.get(b)

    # Return a list of nodes in the graph

    def nodes(self):

        s1 = set([k for k in self.graph\_dict.keys()])

        s2 = set([k2 for v in self.graph\_dict.values() for k2, v2 in v.items()])

        nodes = s1.union(s2)

        return list(nodes)

# This class represent a node

class Node:

    # Initialize the class

    def \_\_init\_\_(self, name:str, parent:str):

        self.name = name

        self.parent = parent

        self.g = 0 # Distance to start node

        self.h = 0 # Distance to goal node

        self.f = 0 # Total cost

    # Compare nodes

    def \_\_eq\_\_(self, other):

        return self.name == other.name

    # Sort nodes

    def \_\_lt\_\_(self, other):

         return self.f < other.f

    # Print node

    def \_\_repr\_\_(self):

        return ('({0},{1})'.format(self.name, self.f))

# A\* search

def astar\_search(graph, heuristics, start, end):

    # Create lists for open nodes and closed nodes

    open = []

    closed = []

    # Create a start node and an goal node

    start\_node = Node(start, None)

    goal\_node = Node(end, None)

    # Add the start node

    open.append(start\_node)

    # Loop until the open list is empty

    while len(open) > 0:

        # Sort the open list to get the node with the lowest cost first

        open.sort()

        # Get the node with the lowest cost

        current\_node = open.pop(0)

        # Add the current node to the closed list

        closed.append(current\_node)

        # Check if we have reached the goal, return the path

        if current\_node == goal\_node:

            path = []

            while current\_node != start\_node:

                path.append(current\_node.name + ': ' + str(current\_node.g))

                current\_node = current\_node.parent

            path.append(start\_node.name + ': ' + str(start\_node.g))

            # Return reversed path

            return path[::-1]

        # Get neighbours

        neighbors = graph.get(current\_node.name)

        # Loop neighbors

        for key, value in neighbors.items():

            # Create a neighbor node

            neighbor = Node(key, current\_node)

            # Check if the neighbor is in the closed list

            if(neighbor in closed):

                continue

            # Calculate full path cost

            neighbor.g = current\_node.g + graph.get(current\_node.name, neighbor.name)

            neighbor.h = heuristics.get(neighbor.name)

            neighbor.f = neighbor.g + neighbor.h

            # Check if neighbor is in open list and if it has a lower f value

            if(add\_to\_open(open, neighbor) == True):

                # Everything is green, add neighbor to open list

                open.append(neighbor)

    # Return None, no path is found

    return None

# Check if a neighbor should be added to open list

def add\_to\_open(open, neighbor):

    for node in open:

        if (neighbor == node and neighbor.f > node.f):

            return False

    return True

# The main entry point for this module

def main():

    # Create a graph

    graph = Graph()

    # Create graph connections (Actual distance)

    graph.connect('Frankfurt', 'Wurzburg', 111)

    graph.connect('Frankfurt', 'Mannheim', 85)

    graph.connect('Wurzburg', 'Nurnberg', 104)

    graph.connect('Wurzburg', 'Stuttgart', 140)

    graph.connect('Wurzburg', 'Ulm', 183)

    graph.connect('Mannheim', 'Nurnberg', 230)

    graph.connect('Mannheim', 'Karlsruhe', 67)

    graph.connect('Karlsruhe', 'Basel', 191)

    graph.connect('Karlsruhe', 'Stuttgart', 64)

    graph.connect('Nurnberg', 'Ulm', 171)

    graph.connect('Nurnberg', 'Munchen', 170)

    graph.connect('Nurnberg', 'Passau', 220)

    graph.connect('Stuttgart', 'Ulm', 107)

    graph.connect('Basel', 'Bern', 91)

    graph.connect('Basel', 'Zurich', 85)

    graph.connect('Bern', 'Zurich', 120)

    graph.connect('Zurich', 'Memmingen', 184)

    graph.connect('Memmingen', 'Ulm', 55)

    graph.connect('Memmingen', 'Munchen', 115)

    graph.connect('Munchen', 'Ulm', 123)

    graph.connect('Munchen', 'Passau', 189)

    graph.connect('Munchen', 'Rosenheim', 59)

    graph.connect('Rosenheim', 'Salzburg', 81)

    graph.connect('Passau', 'Linz', 102)

    graph.connect('Salzburg', 'Linz', 126)

    # Make graph undirected, create symmetric connections

    graph.make\_undirected()

    # Create heuristics (straight-line distance, air-travel distance)

    heuristics = {}

    heuristics['Basel'] = 204

    heuristics['Bern'] = 247

    heuristics['Frankfurt'] = 215

    heuristics['Karlsruhe'] = 137

    heuristics['Linz'] = 318

    heuristics['Mannheim'] = 164

    heuristics['Munchen'] = 120

    heuristics['Memmingen'] = 47

    heuristics['Nurnberg'] = 132

    heuristics['Passau'] = 257

    heuristics['Rosenheim'] = 168

    heuristics['Stuttgart'] = 75

    heuristics['Salzburg'] = 236

    heuristics['Wurzburg'] = 153

    heuristics['Zurich'] = 157

    heuristics['Ulm'] = 0

    # Run the search algorithm

    path = astar\_search(graph, heuristics, 'Frankfurt', 'Ulm')

    print(path)

    print()

# Tell python to run main method

if \_\_name\_\_ == "\_\_main\_\_": main()

### Console Output

['Frankfurt: 0', 'Wurzburg: 111', 'Ulm: 294']

## Genetic Algorithm

# Python3 program to create target string, starting from

# random string using Genetic Algorithm

import random

# Number of individuals in each generation

POPULATION\_SIZE = 100

# Valid genes

GENES = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP

QRSTUVWXYZ 1234567890, .-;:\_!"#%&/()=?@${[]}'''

# Target string to be generated

TARGET = "I love GeeksforGeeks"

class Individual(object):

    '''

    Class representing individual in population

    '''

    def \_\_init\_\_(self, chromosome):

        self.chromosome = chromosome

        self.fitness = self.cal\_fitness()

    @classmethod

    def mutated\_genes(self):

        '''

        create random genes for mutation

        '''

        global GENES

        gene = random.choice(GENES)

        return gene

    @classmethod

    def create\_gnome(self):

        '''

        create chromosome or string of genes

        '''

        global TARGET

        gnome\_len = len(TARGET)

        return [self.mutated\_genes() for \_ in range(gnome\_len)]

    def mate(self, par2):

        '''

        Perform mating and produce new offspring

        '''

        # chromosome for offspring

        child\_chromosome = []

        for gp1, gp2 in zip(self.chromosome, par2.chromosome):

            # random probability

            prob = random.random()

            # if prob is less than 0.45, insert gene

            # from parent 1

            if prob < 0.45:

                child\_chromosome.append(gp1)

            # if prob is between 0.45 and 0.90, insert

            # gene from parent 2

            elif prob < 0.90:

                child\_chromosome.append(gp2)

            # otherwise insert random gene(mutate),

            # for maintaining diversity

            else:

                child\_chromosome.append(self.mutated\_genes())

        # create new Individual(offspring) using

        # generated chromosome for offspring

        return Individual(child\_chromosome)

    def cal\_fitness(self):

        '''

        Calculate fittness score, it is the number of

        characters in string which differ from target

        string.

        '''

        global TARGET

        fitness = 0

        for gs, gt in zip(self.chromosome, TARGET):

            if gs != gt: fitness+= 1

        return fitness

# Driver code

def main():

    global POPULATION\_SIZE

    #current generation

    generation = 1

    found = False

    population = []

    # create initial population

    for \_ in range(POPULATION\_SIZE):

                gnome = Individual.create\_gnome()

                population.append(Individual(gnome))

    while not found:

        # sort the population in increasing order of fitness score

        population = sorted(population, key = lambda x:x.fitness)

        # if the individual having lowest fitness score ie.

        # 0 then we know that we have reached to the target

        # and break the loop

        if population[0].fitness <= 0:

            found = True

            break

        # Otherwise generate new offsprings for new generation

        new\_generation = []

        # Perform Elitism, that mean 10% of fittest population

        # goes to the next generation

        s = int((10\*POPULATION\_SIZE)/100)

        new\_generation.extend(population[:s])

        # From 50% of fittest population, Individuals

        # will mate to produce offspring

        s = int((90\*POPULATION\_SIZE)/100)

        for \_ in range(s):

            parent1 = random.choice(population[:50])

            parent2 = random.choice(population[:50])

            child = parent1.mate(parent2)

            new\_generation.append(child)

        population = new\_generation

        print("Generation: {}\tString: {}\tFitness: {}".\

              format(generation,

              "".join(population[0].chromosome),

              population[0].fitness))

        generation += 1

    print("Generation: {}\tString: {}\tFitness: {}".\

          format(generation,

          "".join(population[0].chromosome),

          population[0].fitness))

if \_\_name\_\_ == '\_\_main\_\_':

    main()

### Console Output

Generation: 1 String: tO{"-?=jH[k8=B4]Oe@} Fitness: 18

Generation: 2 String: tO{"-?=jH[k8=B4]Oe@} Fitness: 18

Generation: 3 String: .#lRWf9k\_Ifslw #O$k\_ Fitness: 17

Generation: 4 String: .-1Rq?9mHqk3Wo]3rek\_ Fitness: 16

Generation: 5 String: .-1Rq?9mHqk3Wo]3rek\_ Fitness: 16

Generation: 6 String: A#ldW) #lIkslw cVek) Fitness: 14

Generation: 7 String: A#ldW) #lIkslw cVek) Fitness: 14

Generation: 8 String: (, o x \_x%Rs=, 6Peek3 Fitness: 13

Generation: 29 String: I lope Geeks#o, Geeks Fitness: 3

Generation: 30 String: I loMe GeeksfoBGeeks Fitness: 2

Generation: 31 String: I love Geeksfo0Geeks Fitness: 1

Generation: 32 String: I love Geeksfo0Geeks Fitness: 1

Generation: 33 String: I love Geeksfo0Geeks Fitness: 1

Generation: 34 String: I love GeeksforGeeks Fitness: 0

## Min-Max and Alpha-Beta pruning

# Python3 program to demonstrate

# working of Alpha-Beta Pruning

# Initial values of Aplha and Beta

MAX, MIN = 1000, -1000

# Returns optimal value for current player

#(Initially called for root and maximizer)

def minimax(depth, nodeIndex, maximizingPlayer,

            values, alpha, beta):

    # Terminating condition. i.e

    # leaf node is reached

    if depth == 3:

        return values[nodeIndex]

    if maximizingPlayer:

        best = MIN

        # Recur for left and right children

        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex \* 2 + i,

                          False, values, alpha, beta)

            best = max(best, val)

            alpha = max(alpha, best)

            # Alpha Beta Pruning

            if beta <= alpha:

                break

        return best

    else:

        best = MAX

        # Recur for left and

        # right children

        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex \* 2 + i,

                            True, values, alpha, beta)

            best = min(best, val)

            beta = min(beta, best)

            # Alpha Beta Pruning

            if beta <= alpha:

                break

        return best

# Driver Code

if \_\_name\_\_ == "\_\_main\_\_":

    values = [3, 5, 6, 9, 1, 2, 0, -1]

    print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))

### Console Output

The optimal value is: 5

## Hill climbing algorithm

# hill climbing search of a one-dimensional objective function

from numpy import asarray

from numpy import arange

from numpy.random import randn

from numpy.random import rand

from numpy.random import seed

from matplotlib import pyplot

# objective function

def objective(x):

    return x[0]\*\*2.0

# hill climbing local search algorithm

def hillclimbing(objective, bounds, n\_iterations, step\_size):

    # generate an initial point

    solution = bounds[:, 0] + rand(len(bounds)) \* (bounds[:, 1] - bounds[:, 0])

    # evaluate the initial point

    solution\_eval = objective(solution)

    # run the hill climb

    solutions = list()

    solutions.append(solution)

    for i in range(n\_iterations):

        # take a step

        candidate = solution + randn(len(bounds)) \* step\_size

        # evaluate candidate point

        candidte\_eval = objective(candidate)

        # check if we should keep the new point

        if candidte\_eval <= solution\_eval:

            # store the new point

            solution, solution\_eval = candidate, candidte\_eval

            # keep track of solutions

            solutions.append(solution)

            # report progress

            print('>%d f(%s) = %.5f' % (i, solution, solution\_eval))

    return [solution, solution\_eval, solutions]

# seed the pseudorandom number generator

seed(5)

# define range for input

bounds = asarray([[-5.0, 5.0]])

# define the total iterations

n\_iterations = 1000

# define the maximum step size

step\_size = 0.1

# perform the hill climbing search

best, score, solutions = hillclimbing(objective, bounds, n\_iterations, step\_size)

print('Done!')

print('f(%s) = %f' % (best, score))

# sample input range uniformly at 0.1 increments

inputs = arange(bounds[0,0], bounds[0,1], 0.1)

# create a line plot of input vs result

pyplot.plot(inputs, [objective([x]) for x in inputs], '--')

# draw a vertical line at the optimal input

pyplot.axvline(x=[0.0], ls='--', color='red')

# plot the sample as black circles

pyplot.plot(solutions, [objective(x) for x in solutions], 'o', color='black')

pyplot.show()

### Console Output

>1 f([-2.74290923]) = 7.52355

>3 f([-2.65873147]) = 7.06885

>4 f([-2.52197291]) = 6.36035

>5 f([-2.46450214]) = 6.07377

>7 f([-2.44740961]) = 5.98981

>9 f([-2.28364676]) = 5.21504

>12 f([-2.19245939]) = 4.80688

>14 f([-2.01001538]) = 4.04016

>15 f([-1.86425287]) = 3.47544

>22 f([-1.79913002]) = 3.23687

>24 f([-1.57525573]) = 2.48143

>25 f([-1.55047719]) = 2.40398

>26 f([-1.51783757]) = 2.30383

>27 f([-1.49118756]) = 2.22364

>28 f([-1.45344116]) = 2.11249

>30 f([-1.33055275]) = 1.77037

>32 f([-1.17805016]) = 1.38780

>33 f([-1.15189314]) = 1.32686

>36 f([-1.03852644]) = 1.07854

>37 f([-0.99135322]) = 0.98278

>38 f([-0.79448984]) = 0.63121

>39 f([-0.69837955]) = 0.48773

>42 f([-0.69317313]) = 0.48049

>46 f([-0.61801423]) = 0.38194

>48 f([-0.48799625]) = 0.23814

>50 f([-0.22149135]) = 0.04906

>54 f([-0.20017144]) = 0.04007

>57 f([-0.15994446]) = 0.02558

>60 f([-0.15492485]) = 0.02400

>61 f([-0.03572481]) = 0.00128

>64 f([-0.03051261]) = 0.00093

>66 f([-0.0074283]) = 0.00006

>78 f([-0.00202357]) = 0.00000

>119 f([0.00128373]) = 0.00000

>120 f([-0.00040911]) = 0.00000

>314 f([-0.00017051]) = 0.00000

Done!

f([-0.00017051]) = 0.000000

### Screenshoots



