Algorithm analysis and design Project

Python package for implementing different algorithm paradigms

Documentation and Report

By,
P Shiridi kumar
2021121005

Package name:"skpalgotools"
Command for installing :
"pip install skpalgotools"

__**Description:**__
The objective/goal of the project is to understand the implementation of the methodologies of different algorithms taught in class and also what not taught in class. The python package developed in the project includes implementation of problems on various algorithm paradigms like dp,greedy ,genetic,graph ..

__**Outcome:**__
In this journey of doing the project i not just learned theoretically in the class but also came out of my comfort zone and implemented them in practical which made me understand the concepts better and clear all minor doubts .At the end i can say i have a good satisfaction on my project and also improved my understanding algorithm paradigms and coding skills(implementation). Also I have learned new concepts like genetic algorithms and other optimization algorithms(hill climbing but was not able to implement them).

Note: Definitions for various problem statements are taken from different resources available on the net for the documentation purpose.

__**Declaration:**__

I hereby declare that the entire code for implementing the package library functions are done solely by me and did not copy from any external resources
****

Documentation of Topics covered(and learned) in the project

Travelling salesman problem optimization(Implemented using genetic algorithm)**:**
Problem:Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.
There is no polynomial time solution available for this problem as the problem is a known NP-Hard problem. There are approximate algorithms to solve the problem though.

So i have implemented and tried to optimize the travelling salesman problem using genetic algorithms

Genetic algorithms:
Genetic algorithms are evolutionary algorithms used or optimization problems(Discrete as well as continuous).
For travelling salesman problem we are using genetic algorithms and we need to optimize the cost of the cities to be covered .We need to minimise the cost. Cost here is the distance taken to cover all the cities (exactly once)and return to the starting point.

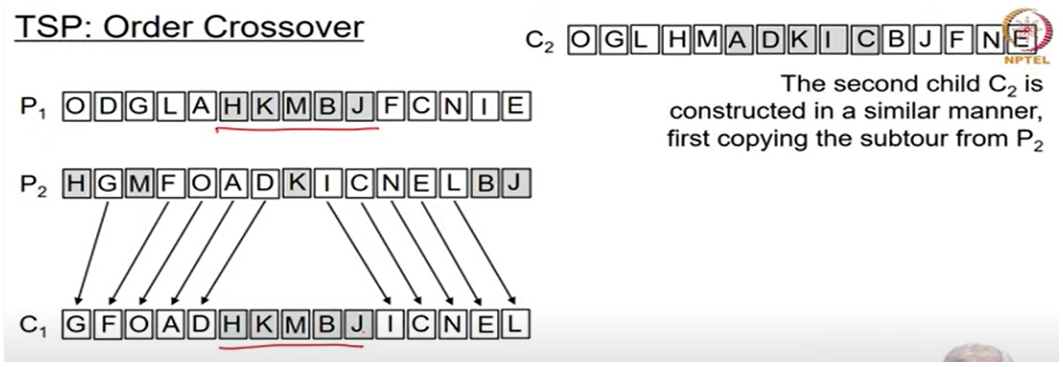
Terms we need to define for the genetic Algorithm:
State: A Sample of permutation of cities .

Cost function/Fitness function: In this problem ,the cost is the distance taken to cover all the cities and return to the starting point.

Neighbourhood function: In this problem ,we define the neighbourhood function to be the permutation of the cities which differs by swapping of two cities from the actual permutation

Mutation:
Swapping of two cities in the current permutation

Crossover:
In this problem we used order crossover to ensure that there are no 2 cities repeated in a sample.


TSP: Order Crossover
C_2 O G L H M A D K I C B J F N E
NPTEL
The second child C_2 is constructed in a similar manner, first copying the subtour from P_2
P_1 O D G L A H K M B J F C N I E
P_2 H G M F O A D K I C N E L B J
C_1 G F O A D H K M B J I C N E L
(Resource: NPTEL Slides)

Population:
Initial states which could be randomly generated.

Parameters:

Weights: corresponds to the adjacency matrix with the distances between the cities(if a city is not reachable then it is assigned as 0 distance)

n: represents the number of cities

Population: initial number states to be populated randomly(default value is 100)

Mutation_prob: at each iteration the state is mutated with a prob value of mutation prob.(default value =0.5)

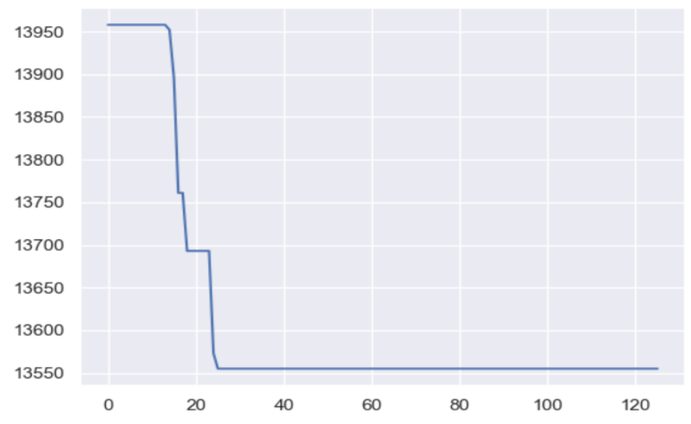
Max_iter : maximum number of iterations in a plateau(default value =100)

Sample code:

```
import skpalgotools
from skpalgotools import GeneticAlgo
obj=GeneticAlgo(n,weights,population=1000,mutation_prob=0.4,max_iter=100)
state,cost=obj.solve()
print(state,cost)
```

output

(The function returns best possible state found and also the fitness value and also the fitness function graph as shown below)



(sample output graph)

Coin change problem(Dynamic programming):
Problem statement: Given a value N, find the number of ways to make change for N cents, if we have infinite supply of each of S = { S1, S2, .. , SM } valued coins.

Key idea: At each step we have two choices we can either take the coin and decrement the value for n and find the solution for n-coin_value or we don't take the coin and move to the next coin and find the solution again for n we increment the solutions to 1 when n reaches 0 at each step.And this recurrence relation(have overlapping sub problems) can be in turn converted into a dp table from bottom up approach.

Function: **coinchange(arr,m,n)**
Parameters:
arr=contains the coins values
m=no of distinct coins values
n= target value

Sample code:

```
import skpalgotools
from skpalgotools import algos
a=algos.coinchange([2,5,3,6],4,10)
print(a)
```

Sample output:
5

Knapsack problem:

given weights and values of N items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. Note that we have only one quantity of each item.
In other words, given two integer arrays val[0..N-1] and wt[0..N-1] which represent values and weights associated with N items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item or don't pick it (0-1 property).{**problem definition source** :geeks for geeks}

Key idea:
At each step we can either take the item and decrease our capacity and increase our value by value of that item or go to the next item and don't decrease the capacity. And this recurrence relation causes many overlapping subproblems which in turn can be memorised or we can create dp table from bottom up approach.

Function: **knapSack(Capacity,wts,values,num_items)**

Parameters:
capacity:Maximum weight capacity of the bag
Wts: weights of the items
Values : profit values of the item
Num_items: number of items

Sample code:

```
import skpalgotools
from skpalgotools import algos
b=algos.knapSack(4,[4,5,1],[1,2,3],3)
print(b)
```

Sample output :

```
3
```

Matrix chain multiplication:

Problem: Given a sequence of matrices, find the most efficient way to multiply these matrices together. The efficient way is the one that involves the least number of multiplications.(**problem definition source : Geeks for geeks)**

Key idea: The problem here is to find the best associativity . So we can divide the problem into subproblems and combine them (and this turns out to be combinatorial). So we start from the base solution and form a dp table.

Function:**matrixMultiplication(N, arr):**

The dimensions of the matrices are given in an array arr[] of size N (such that N = number of matrices + 1) where the ith matrix has the dimensions (arr[i-1] x arr[i]).(Format source :Geeks for geeks)
Parameters:
N=number of matrices+1
arr=matrix dimensions

Sample code:

```
import skpalgotools
from skpalgotools import algos
b=algos.matrixMultiplication(5, [40,20,30,10,30])
print(b)
```

Sample output:
```
26000
```

Finding N chooses R(nCr):

Problem: Given n and r we need to find the number of combinations to r out of n.

Key idea: At each step we can choose the 1 form r and decrement value of r and n or decrement the value of r and not decrement value of n.(A simple dp table can be constructed)

Function:**nCr(n, r)**

Where n is the total number and we need to choose r from it.

Sample code:

```
import skpalgotools
```

```
from skpalgotools import algos
b=algos.nCr( 8,2)
print(b)
```

Sample output:

```
28
```

editDistance:

Problem:
Given two strings s and t. Return the minimum number of operations required to convert s to t.
The possible operations are permitted:

Insert a character at any position of the string.
Remove any character from the string.
Replace any character from the string with any other character.
(Problem definition source: Geeks for geeks)

Key idea: At each step we can actually do 3 things either remove a char from first word1 and increase the distance by one and find the edit distance for the resulting strings and the same for wrd2 applies and the last thing is we can remove(equivalent to replace a character)(if char in both the words are diff) and increase distance by 1/move(if char in both the words are different) and distance remains by same. We need to find minimum of all the above sub parts(they are overlapping ,so we start from the base solution and form a dp table)

Function:**editDistance(s, t):**

Parameters:
S: The first string
T: The second string

Sample code:

```
import skpalgotools
from skpalgotools import algos
```

```
b= algos.editDistance("shiridi", "thrdi")
print(b)
```

Sample output:
3(we can replace s with t and remove 2 i’s)

equalPartition:

Problem:Given an array arr[] of size N, check if it can be partitioned into two parts such that the sum of elements in both parts is the same.

Key idea:
We can the model the problem as , finding a set of elements in arr which sum to sum(arr)//2 .To do this in each step we have 2 options i.e, include the element of the arr and increase sum to the value of the element or don’t include the element and go to the next.(We can form a simple dp starting from the base solution).

Function:**equalPartition(N, arr):**

Parameters:
N:The number of elements in the array
Arr: input array

Sample code:

```
import skpalgotools
from skpalgotools import algos
b= algos.equalPartition(5, [10,6,6,3,7])
print(b)
```

Sample output:
True

Longest common subsequence:

Problem: Given two strings ,we need to find the longest common subsequence of the two strings.

Key idea:
We can start from the last characters for both the strings. At each step,if the corresponding char are not equal we can have two options remove the remove the last char of the string1 and find lcs for the resulting strings and follows the same for string2 (we need to find the max of them) and if they are equal we increase the count by 1, Return count.

Function:**lcs(self,x,y,str1,str2)**
Parameters:
X,y :length of str1 and str2
Str1,str2 : string 1 and string 2

Sample code:

```
import skpalgotools
from skpalgotools import algos
b= algos.lcs(7,6,"shiridikumar","akanksha")
print(b)
```

Sample output:
3

Maximize the cut segments:

Problem:
Given an integer N denoting the Length of a line segment. You need to cut the line segment in such a way that the cut length of a line segment each time is either x , y or z. Here x, y, and z are integers.After performing all the cut operations, your total number of cut segments must be maximum.(problem definition source: Geeks for geeks)

Key idea:
At each step we have three options: we can cut a segment of x or y or z and we can decrease the available length accordingly and again increase the segment by 1 if the solution tends to exactly cut the n into different segments and recurse the solution(take a maximum of three possible sub parts) and return no of segments .(as we can see the problem can be simply constructed using 1d dp array)

Function:**maximizeTheCuts(n,x,y,z):**

Parameters:
N the length of the line segment
X,y,z: length of each segment sizes

Sample code:

```
import skpalgotools
from skpalgotools import algos
b= algos.maximizeTheCuts(4,2,1,1)
print(b)
```

Sample output:
4

Longest repeating subsequence

Problem**:**Given a string str, find the length of the longest repeating subsequence such that it can be found twice in the given string. The two identified subsequences A and B can use the same ith character from string str if and only if that ith character has different indices in A and B.(Problem definition source :Geeks for geeks)

Key idea:
The problem can be modelled similarly to Longest common subsequence between two same strings with only a difference that the subsequence should not start from the same index.

Function:**LongestRepeatingSubsequence(str):**

Parameters:
Str: Input String

Sample code:

```
import skpalgotools
from skpalgotools import algos
b= algos.LongestRepeatingSubsequence("ahshaoashddfnkdjsvs")
```

```
print(b)
```

Sample output:
6

Longest increasing subsequence:

Problem:Given sequence of integers, we need to find the longest increasing subsequence from the sequence.

Key idea:Each number can form a subsequence of length 1 by just including itself . We start by adding each element one by one . when a new number is added its solution will be the maximum of the solutions of the numbers which are less than 1.

Function**:longestSubsequence(a,n)**

Parameters:

A:sequence of integers
N:length of the sequence

Sample Code:

```
import skpalgotools
from skpalgotools import algos
b= algos.longestSubsequence([0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15],16)
print(b)
```

Sample output:
6

largest_subarray_sum(Kadane's algorithm)
Problem:To find the largest contiguous subarray sum.

Key idea:
We maintain two variables temp and curr, we will add each element one by one to temp .id its resulting to <0 we will make it 0 if it's greater than curr we will change curr to temp.

Function:**largest_subarray_sum(a,n)**

Parameters:
A: array of integers
N: length of the array

Sample code:

```
import skpalgotools
from skpalgotools import algos
e=algos.largest_subarray_sum([1,2,3-2,5],5)
print(e)
```

Sample output:
9

Dijkstra's algorithm (single source shortest path algorithm)

Problem: Given a graph, we need to find the shortest path to all the vertices from a source vertex.

Key idea:
We need to initially initialise all the node distances to be infinite and source vertex to 0 . We include the source vertex in the explored list. We will now pick the vertex with least distance (source 0)We will now update the distances from the picked vertex to all the vertices which are adjacent vertices of the picked vertex . We update the distances of the adjacent vertices by comparing their distance from source and distance[picked]+cost(picked,adjacent) we take the minimum of the both. And then we pick the vertex and the procedure repeats.

Function:dijikstra(num_nodes ,edges ,source)

Parameters
num_nodes:No of nodes

Edges: list of (u,v,cost) sequences.{where u and v nodes and cost is the weight of the edge}
source:Source node

Sample code:

```
import skpalgotools
from skpalgotools import algos
b= dijikstra(4 ,[[1, 2, 24], [1, 4, 20], [3, 1, 3], [4, 3, 12]] ,1)
print(b)
```

Sample output:
[24, 3, 15](distances from source to all other vertices)

Prims algorithm:
Problem: Given a graph , we need to find the minimum cost spanning tree.

Key idea:
Prims algorithm is a greedy idea.We start with any source vertex and then we include it to the explored list and then explore its neighbours and choose the one with minimum distance from the current available vertex in the explored list and also it should not form a cycle and the process repeats.

Function:prims(num_nodes ,edges ,source)

Parameters
num_nodes:No of nodes
Edges: list of (u,v,cost) sequences.{where u and v nodes and cost is the weight of the edge}
source:Source node

Sample code:

```
import skpalgotools
from skpalgotools import algos
b= prims(4 ,[[1, 2, 24], [1, 4, 20], [3, 1, 3], [4, 3, 12]] ,1)
print(b)
```

Sample output:
39

Breadth first search:

Function:bfs(n,edges,source)

Parameters:
N=number of edges
edges=List of (u,v) pairs where u,v are the nodes and each pair indicates an edge between them
source=source vertex

Key idea:
Queue implementation

Sample code:

```
from skpalgotools import algos
a=algos.bfs(4 ,[[1, 2], [2, 4], [3, 1], [4, 3]], 1)
print(a)
```

Sample output:
[1, 2, 3, 4]

depth first search:

Function:dfs(n,edges,source)

Parameters:
N=number of edges
edges=List of (u,v) pairs where u,v are the nodes and each pair indicates an edge between them
source=source vertex

Key idea:
Recursive implementation

Sample code:

```
from skpalgotools import algos
a=algos.dfs(4 ,[[1, 2], [2, 4], [3, 1], [4, 3]], 1)
print(a)
```

Sample output:
[1, 2, 4, 3]

Diameter of a binary tree:
Problem:Given a binary tree we need to find the diameter of the binary tree(the longest path in the tree)

Function:**diameter_tree(root)**
Parameters:
root:Pointer to the root node

Sample code implementation:

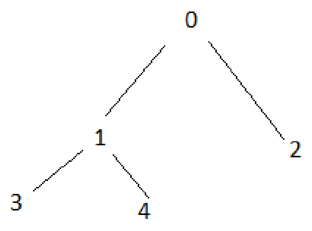
```
import skpalgotools
class node:
    def __init__(self,val):
        self.val=val
        self.right=None
        self.left=None

from skpalgotools import diameter_tree
n=int(input())
g=[node(0)]
for i in range(n):
    g.append(node(i+1))
for i in range(n):
    u,v,w=map(int,input().split())
    if(w!=-1):
        g[u].right=g[w]
    if(v!=-1):
        g[u].left=g[v]

a=diameter_tree(g[0])
print(a)
```

Sample input:
5
0 1 2
1 3 4

2 -1 -1
3 -1 -1
4 -1 -1

Sample output:
3

(input format 1st line is number of nodes ,next n lines: node number,node left child,node right child, if no child then -1)



(Above input graph representation)

Binary_tree_height

Problem:Given a binary tree we need to find its height

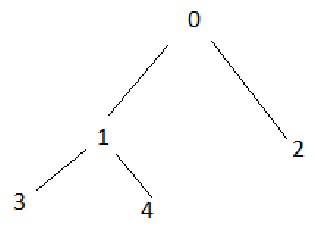
Function:**binary_tree_height(root)**
Parameters:
root:Pointer to the root node

Sample code implementation:

```
import skpalgotools
class node:
    def __init__(self,val):
        self.val=val
        self.right=None
        self.left=None

from skpalgotools import binary_tree_height
n=int(input())
g=[node(0)]
```

```
for i in range(n):
    g.append(node(i+1))
for i in range(n):
    u,v,w=map(int,input().split())
    if(w!=-1):
        g[u].right=g[w]
    if(v!=-1):
        g[u].left=g[v]

a=binary_tree_height(g[0])
print(a)
```

(input format 1st line is number of nodes ,next n lines: node number,node left child,node right child, if no child then -1)



(Above input graph representation)

Sample output:
3

Kthrank element in an array:
Problem:GIven an array of elements and a value k ,we need to find the kth rank(kth smallest) element in array

Key idea:
We will use a divide and conquer strategy. We divide the array of elements into groups of 5 elements and find the median of all the groups and then median of these medians. We now use this median to partition the array(we move all the elements whose values are < median to the left and the others to right) after partitioning if the medians position is at k we return median if its less than k we find the solution at the right partition and if its greater than k we find the solution again in the left partition.

Function:**kthrank(arr,n,k)**
Parameters:
Arr: array of integers
n:size of the array
k: k value

Sample code:

```
from skpalgotools import algos
a=algos.kthrank([4,20,7,3,6,10],6,3)
print(a)
```

Sample output:
6

Thank You