Dynamic Programming, Graph Algorithm Problems and Their Solutions In Code

compiled by D. Gueorguiev, 8/21/2024

Table of Contents

[Problems Solvable by Dynamic Programming 1](#_Toc176628651)

[The N-th Fibonacci Number 1](#_Toc176628652)

[Rod Cutting Problem 2](#_Toc176628653)

[Min Cost Path 2](#_Toc176628654)

[Moving in An Array in an Optimal Way (Denardo) 4](#_Toc176628655)

[Weighted Job Scheduling 4](#_Toc176628656)

[Longest Common Sequence 6](#_Toc176628657)

[0/1 Knapsack Problem 7](#_Toc176628658)

[Floyd-Warshall Algorithm 8](#_Toc176628659)

[Bellman-Ford Algorithm 8](#_Toc176628660)

[Vertex Cover Problem 8](#_Toc176628661)

[Travelling Salesman Problem 8](#_Toc176628662)

[Longest Palindromic Subsequence 8](#_Toc176628663)

[Minimum Insertions to Form a Palindrome 8](#_Toc176628664)

[Matrix Chain Multiplication 8](#_Toc176628665)

[Problems Solvable By Recursive Backtracking 9](#_Toc176628666)

[N Queens Problem 9](#_Toc176628667)

[2 x 2 Sudoku Solver 9](#_Toc176628668)

[Related Graph Algorithms 9](#_Toc176628669)

[Dijkstra’s Shortest and Longest Path Algorithms 9](#_Toc176628670)

[Breath-First Search Algorithm 11](#_Toc176628671)

[Depth-First Search Algorithm 11](#_Toc176628672)

[References 12](#_Toc176628673)

# Problems Solvable by Dynamic Programming

## The N-th Fibonacci Number

//TODO: Finish the section on the N-th Fibonacci number

## Rod Cutting Problem

Given a rod of length inches (or centimeters) and a table of prices for find the max obtainable revenue by cutting the rod and selling the pieces.

The problem must exhibit optimal substructure if in order to solve the original problem of size we solve similar problems of the same type but smaller sizes.

(1)

Execution time complexity:

(2)

The solution of (2) is given by

(3)

(3) represents exponential time complexity.

Let us assume we are given an integer n and a list p with len(p) >= n. We want to find the max revenue and a partition of n which achieves it.

in code:

def cut\_rod(n: int, p: list[int]):

if n == 0:

return 0

//TODO: Finish the section on the rod cutting problem

## Min Cost Path

Given a cost matrix cost[][] and a position (M,N) in cost[][], write a function that returns cost of minimum cost path to reach (M,N) from (0,0). Each cell of the matrix represents a cost to traverse through that cell. The total cost of a path to reach (M,N) is the sum of all the costs on that path (including both source and destination). You can only traverse down , right, and diagonally lower cells from a given cell, i.e. from a given cell (i,j), cells (i+1,j), (i,j+1), and (i+1,j+1) can be traversed.

Example:

1, 2, 3

4, 8, 2

1, 5, 3

The path with min cost is (0,0)->(0,1)->(1,2)->(2,2). The cost of the path is 1+2+2+3=8.

Naïve implementation of the solution using recursive algorithm

import sys

def minCost(cost, m, n):

if n < 0 or m < 0:

return sys.maxsize

elif m == 0 and n == 0:

return cost[m][n]

else:

return cost[m][n] + min(minCost(cost, m-1, n-1),

minCost(cost, m-1, n),

minCost(cost, m, n-1))

Slightly better implementation using recursive algorithm with memoization

import sys

def minCostMemoized(cost, m, n):

if n < 0 or m < 0:

return sys.maxsize

elif m == 0 and n == 0:

return cost[m][n]

if memo[m][n] != -1:

return memo[m][n]

memo[m][n] = cost[m][n] + min(

minCostMemoized(cost, m-1, n-1),

minCostMemoized(cost, m-1, n),

minCostMemoized(cost, m, n-1))

Here memo is a list of lists with sizes equal to the row length and col length of the cost matrix initialized originally to -1:

memo = [[-1] \* C for \_ in range(R)] # R = len(cost) and C = len(cost[0])

Dynamic Programming formulation and implementation of the min cost problem:

minCost(1,1)

minCost(0,0)

minCost(0,1)

minCost(1,0)

minCost(1,2)

minCost(0,1)

minCost(0,2)

minCost(1,1)

minCost(2,1)

minCost(1,0)

minCost(1,1)

minCost(2,0)

minCost(2,2)

def minCost(cost, m, n):

tot\_cost = [[0 for x in range(n+1)] for x in range(m+1)]

tot\_cost[0][0] = cost[0][0]

# Initialize first column of tot\_cost array

for i in range(1, m+1):

tot\_cost[i][0] = tot\_cost[i-1][0] + cost[i][0]

# Initialize first row of tot\_cost array

for j in range(1, n+1):

tot\_cost[0][j] = tot\_cost[0][j-1] + cost[0][j]

# Construct rest of the tot\_cost array

for i in range(1, m+1):

for j in range(1, n+1):

tot\_cost[i][j] = min(tot\_cost[i-1][j-1], tot\_cost[i-1][j], tot\_cost[i][j-1]) + cost[i][j]

return tot\_cost[m][n]

Dynamic Programming formulation without the additional space complexity of tot\_cost

def minCost(cost, row, col):

# For 1st column

for i in range(1, row):

cost[i][0] += cost[i - 1][0]

# For 1st row

for j in range(1, col):

cost[0][j] += cost[0][j - 1]

# For rest of the 2d matrix

for i in range(1, row):

for j in range(1, col):

cost[i][j] += (min(cost[i - 1][j - 1],

min(cost[i - 1][j],

cost[i][j - 1])))

# Returning the value in

# last cell

return cost[row - 1][col - 1]

//TODO: Finish the section on the Min Cost Path problem

## Moving in An Array in an Optimal Way (Denardo)

//TODO: Finish the section on the problem of Moving In An Array in an Optimal Way

## Weighted Job Scheduling

Given are N jobs where each job is represented by following three elements of it.

1 ) Start Time

2 ) Finish Time

3 ) Profit / Associated Value (>= 0)

Find the maximum profit subset of jobs such that no two jobs in the subset overlap

Example:

Input:

Number of jobs n = 4

Job Details {Start Time, Finish Time, Profit}

Job 1: {1, 2, 50}

Job 2: {3, 5, 20}

Job 3: {6, 19, 100}

Job 4: {2, 100, 200}

Output:

Max profit is 250

Max profit can be attained by scheduling jobs 1 and 4.

Recursive solution for the job scheduling problem:

1 ) Sort jobs according to finish time

2 ) Apply the following recursive process

# here `arr` is array of n jobs

def findMaximumProfit(arr, n):

a) if n == 1 return arr[0]

b) return the maximum of the following two profits

(i) maximum profit by excluding current job : findMaximumProfit(arr, n-1)

(ii) maximum profit including the current job

How to find the profit including the current job?

The idea is to find the latest job before the current job in the sorted array that does not conflict with the current job arr[n-1]. Once we find such a job, we recur for all jobs until that job and add profit of current job to result.

Naïve recursion approach:

from functools import cmp\_to\_key

# A job has start time, finish time and profit

class Job:

def \_\_init\_\_(self, start, finish, profit):

self.start = start

self.finish = finish

self.profit = profit

def jobComparator(s1, s2):

return s1.finish < s2.finish

# Find the latest job (in sorted array) that

# doesn't conflict with the job[i]. If there

# is no compatible job, then it returns -1

def latestNonConflict(arr, i):

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

if arr[j].finish <= arr[i - 1].start:

return j

return -1

# A recursive function that returns the

# maximum possible profit from given

# array of jobs. The array of jobs must

# be sorted according to finish time

def findMaxProfitRec(arr, n):

# Base case

if n == 1:

return arr[n - 1].profit

# Find profit when current job is included

inclProf = arr[n - 1].profit

i = latestNonConflict(arr, n)

if i != -1:

inclProf += findMaxProfitRec(arr, i + 1)

# Find profit when current job is excluded

exclProf = findMaxProfitRec(arr, n - 1)

return max(inclProf, exclProf)

# The main function that returns the maximum

# possible profit from given array of jobs

def findMaxProfit(arr, n):

# Sort jobs according to finish time

arr = sorted(arr, key = cmp\_to\_key(jobComparator))

return findMaxProfitRec(arr, n)

Dynamic Programming Implementation

Uses the earlier defined class Job, jobComparator and latestNonConflict functions

def findMaxProfit(arr, n):

# Sort jobs according to finish time

arr = sorted(arr, key=cmp\_to\_key(jobComparator))

# Create an array to store solutions of subproblems.

# table[i] stores the profit for jobs till arr[i]

# (including arr[i])

table = [None] \* n

table[0] = arr[0].profit

# Fill entries in M[] using recursive property

for i in range(1, n):

# Find profit including the current job

inclProf = arr[i].profit

l = latestNonConflict(arr, i)

if l != -1:

inclProf += table[l]

# Store maximum of including and excluding

table[i] = max(inclProf, table[i - 1])

# Store result and free dynamic memory

# allocated for table[]

result = table[n - 1]

return result

## Longest Common Sequence

Let us consider strings in the form where are strings or characters. We say that the tuple forms a sequence in if appears on the left of for each pair such that . Given two strings and find the length of the largest common sequence .

#### Naïve Implementation

def longest\_common\_sequence(s1: str, s2: str, m: int, n: int):

“““

Determine the length of the longest common sequence.

:param s1: str

:param s2: str

:param m: int

:param n: int

“““

if m == 0 or n == 0:

return 0

elif s1[m-1] == s2[n-1]:

return 1 + longest\_common\_sequence(s1, s2, m-1, n-1)

else:

return max(longest\_common\_sequence(s1, s2, m, n-1),

longest\_common\_sequence(s1, s2, m-1, n))

**Excerpt 7**: naïve implementation for the longest common sequence

#### Dynamic Programming implementation

Optimal Subproblem formulation

Initially,

//TODO: Finish the section on the LCS problem

## 0/1 Knapsack Problem

Given are items where each item has some weight and profit associated with it. Also, it is given a bag with capacity - that is, the bag can hold at most weight in it. The task is to put such combination of items in the bag so that the profit is maximized. The constraint is that we can put an item in the bag, or we cannot put it at all, it is not possible to put only a fraction of it.

Input: N = 3, W = 4, profit = [1, 3, 4], weight = [4, 5, 1]

Let us denote with

Naïve recursive algorithm:

# A naive recursive implementation

# of 0-1 Knapsack Problem

def knapSack(W, wt, val, n):

# Base Case

if n == 0 or W == 0:

return 0

# If weight of the nth item is

# more than Knapsack of capacity W,

# then this item cannot be included

# in the optimal solution

if (wt[n-1] > W):

return knapSack(W, wt, val, n-1)

# return the maximum of two cases:

# (1) nth item included

# (2) not included

else:

return max(

val[n-1] + knapSack(

W-wt[n-1], wt, val, n-1),

knapSack(W, wt, val, n-1))

1 ) find the subset of elements , each with a weight less than . That is,

2 )

//TODO: Finish the section on the 0/1 Knapsack problem

## Floyd-Warshall Algorithm

//TODO: Finish the section on the Floyd-Warshall algorithm

## Bellman-Ford Algorithm

//TODO: Finish the section on the Bellman-Ford algorithm

## Vertex Cover Problem

//TODO: Finish the section on the Vertex Cover problem

## Travelling Salesman Problem

//TODO: Finish the section on the TSP problem

## Longest Palindromic Subsequence

//TODO: Finish the section on the Longest Palindromic Subsequence problem

## Minimum Insertions to Form a Palindrome

//TODO: Finish the section on the Min Inserts to Form A Palindrome problem

## Matrix Chain Multiplication

//TODO: Finish the section on the Matrix Chain Multiplication

# Problems Solvable By Recursive Backtracking

## N Queens Problem

Place queens on board so that no pair of Queens attack each other.

For example, when N = 4 we have:

.Q..

...Q

Q...

..Q.

//TODO: Finish the section on the N Queens problem

## 2 x 2 Sudoku Solver

//TODO: Finish the section on the 2 x 2 sudoku solver algorithm

# Related Graph Algorithms

## Dijkstra’s Shortest and Longest Path Algorithms

### Shortest path algorithm with undirected graphs

Let us consider first the case of undirected graphs

Given a weighted undirected graph with its set of vertices and set of arcs and a source vertex in the graph , find the shortest paths from to all other vertices , in . The graph does not contain any negative edge. For simplicity, we will denote all nodes in the graph with their indices only e.g. and instead of and .

0

1

2

3

4

7

6

5

8

4

8

7

11

7

6

1

4

14

9

10

2

8

2

Figure : Weighted undirected graph

Implementation of Dijsktra’s algorithm using *Iteratively Expanding Distance Front*

The idea is to generate SPT with a given target vertex . We maintain an [adjacency matrix](https://en.wikipedia.org/wiki/Adjacency_matrix) with two sets:

- one set contains vertices included in the shortest-path tree

- other set includes vertices not yet included in the SPT

At every step of the algorithm, find a vertex that is in the other set (set not yet included) and has a minimum distance from source.

- create a shortest path tree set spt\_set that keeps track of vertices included in the SPT i.e. those vertices which min distance to has been calculated. Initially is spt\_set empty.

- assign a distance value to all vertices in the graph. Initialize all distance values as sys.maxsize. Assign the distance value of 0 for vertex so that it is picked first.

- while spt\_set does not include all vertices

> pick a vertex that is not in spt\_set and has a minimum distance value

> include to spt\_set

> update the distance value of all adjacent vertices of

- to update the distance values, iterate through all adjacent vertices

- for every adjacent vertex , if the sum of the distance value to from and length of edge is less than the distance value of i , then update the distance value of .

Note: we use a Boolean array spt\_set[] to represent the set of vertices included in SPT. If a value spt\_set[i] is true, then vertex is included in SPT, otherwise not. Array dist[] is used to store the shortest distance values of all vertices.

import sys

class DijkstraIEFD:

“””

Iteratively Expanding Front of Distances Algorithm (IEFD)

“””

def \_\_init\_\_(self, vertex\_count, adjacency\_matrix):

self.vertex\_count = vertex\_count

self.adjacency\_matrix = adjacency\_matrix

# A utility function to find the vertex with

# minimum distance value, from the set of vertices

# not yet included in shortest path tree

def min\_distance(self, dist, spt\_set):

# Initialize minimum distance for next node

min\_dist = sys.maxsize

# Search nearest vertex not in the shortest path tree

for k in range(self.vertice\_count):

if dist[k] < min\_dist and spt\_set[k] is False:

min\_dist = dist[k]

min\_index = k

return min\_index

# Function that implements Dijkstra's single source

# shortest path algorithm for a graph represented

# using adjacency matrix representation

def run(self, s):

dist = [sys.maxsize] \* self.vertex\_count

dist[s] = 0

spt\_set = [False] \* self.vertex\_count

for \_ in range(self.vertex\_count):

# Pick the minimum distance vertex from

# the set of vertices not yet processed.

# x is always equal to t in first iteration

x = self.min\_distance(dist, spt\_set)

# Put the minimum distance vertex in the

# shortest path tree

spt\_set[x] = True

# Update dist value of the adjacent vertices

# of the picked vertex only if the current

# distance is greater than new distance and

# the vertex in not in the shortest path tree

for y in range(self.vertex\_count):

if self.graph[x][y] > 0 and spt\_set[y] is False and \

dist[y] > dist[x] + self.graph[x][y]:

dist[y] = dist[x] + self.graph[x][y]

return dist

Example of adjacency matrix for the undirected graph on the Figure above:

adj\_matr = [[0, 4, 0, 0, 0, 0, 0, 8, 0],

[4, 0, 8, 0, 0, 0, 0, 11, 0],

[0, 8, 0, 7, 0, 4, 0, 0, 2],

[0, 0, 7, 0, 9, 14, 0, 0, 0],

[0, 0, 0, 9, 0, 10, 0, 0, 0],

[0, 0, 4, 14, 10, 0, 2, 0, 0],

[0, 0, 0, 0, 0, 2, 0, 1, 6],

[8, 11, 0, 0, 0, 0, 1, 0, 7],

[0, 0, 2, 0, 0, 0, 6, 7, 0]]

For undirected graphs obviously adj\_matr[i][j] = adj\_matr[j][i] .

### Shortest path in case of directed weighted graphs

The undirected graph from the previous Figure transforms into:

0

1

2

7

8

3

5

6

4

4

8

7

11

7

6

1

4

14

9

10

2

8

2

Figure : Weighted directed graph

Note that the topology of is the same as that of which means that the latter is the same as the former up to isomorphism sans the directionality. Intuitively, this means that each node in first graph can be mapped

The adjacency matrix of the directed graph (digraph) on the Figure above is :

Implementation of Dijkstra’s Algorithm using Dynamic Programming

We have two special vertices – source vertex and target vertex which will be indexed with and accordingly. We want to find the min distance path from to .

Let us denote with the minimum distance from vertex to the target vertex .

Let us denote with the length of the arc .

Then we have the following inequality for every pair of vertices

, (1)

The shortest path from node to node obviously satisifies:

(2)

Using (2) we can generate the shortest path tree (SPT) to vertex from every other vertex in the graph.

**Longest Path Algorithm and Tree of Longest Paths**

Now we denote with the length of the longest path from the source node to node .

Then the following inequality holds:

(3)

Note that represents the length of *a* path from the source node to node .

Then from all paths to node in the form the longest path must be given with:

(4)

Critical Path

Reinterpret the semantics of the arcs values in the directed graphs (e.g.

Related problem : Find the

//TODO: Finish the section on the Dijkstra’s Shortest Path algorithm implementations

## Breath-First Search Algorithm

//TODO: Finish the section on the Breath-First Search algorithm

## Depth-First Search Algorithm

//TODO: Finish the section on the Depth-First Search algorithm

# References

[GeeksForGeeks Dynamic Programming Intro](https://www.geeksforgeeks.org/dynamic-programming/)

[Dynamic Programming: Models and Applications, Eric Denardo, 1982, Yale U.](https://github.com/dimitarpg13/graphs_and_dynamic_programming/blob/master/books/dynamic-programming-models-and-application-denardo.pdf)

[Dynamic Programming, Richard Bellman, Princeton, 1957, Sixth Print 1972](https://github.com/dimitarpg13/graphs_and_dynamic_programming/blob/master/books/1957-bellman-dynamicprogramming.pdf)

[Digraphs Theory, Algorithms, and Applications, Bang-Jensen, and Gutin, 2007](https://github.com/dimitarpg13/graphs_and_dynamic_programming/blob/master/books/DigraphsJensenGutin.pdf)