Folha 8 – Otimização

Exercícios

1.
(a) Descreva o algoritmo de força bruta que encontre a solução óptima para o problema da mochila 0/1.

(b) Qual a complexidade em termos de O-grande do algoritmo descrito, em função do número de itens disponíveis? Discuta a viabilidade deste algoritmo.

(c) Se a coleção de partida a considerar tiver 100 itens e cada operação básica deste algoritmo levar 1 nanosegundo a ser executada, quanto tempo (em anos) seria preciso esperar pela solução para este caso.

2.
Considere agora o seguinte algoritmo guloso (*greedy*) para a solução do problema da mochila 0/1:

1. Começar com a mochila vazia
2. Ordenar os itens por valor
3. Percorrer os itens do maior para o menor valor:
   3.1. Se o item couber na mochila, adiciona-o.

(a) Qual a complexidade em termos de O-grande deste algoritmo?

(b) Se a coleção de partida a considerar tiver 100 itens e cada operação básica deste algoritmo levar 1 nanosegundo a ser executada, quanto tempo (em segundos) seria preciso esperar pela solução para este caso.

(c) Teste o algoritmo com os seguintes itens e capacidade da mochila = 10 Kg:

TV	20Kg	200€
Microondas	9Kg	50€
Computador	5Kg	40€
Aspirador	4Kg	35€
Livros	1 Kg	5€

A solução encontrada pelo algoritmo é a melhor? Como poderia alterar o algoritmo para melhorar a solução encontrada? Volte a testar o seu algoritmo melhorado com os mesmos itens.

(d) Considere que para itens de valor semelhante, são preferidos aqueles que têm nomes mais curtos. Como se pode alterar o algoritmo para dar preferência a estes itens, mantendo o objetivo de alcançar uma mochila com o máximo valor possível?

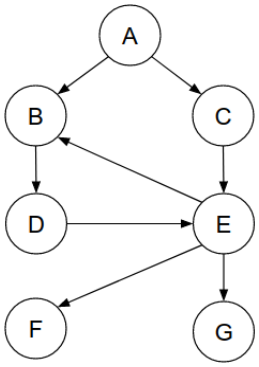
3.
Considere a implementação deste algoritmo no Anexo A. Considere que só se consegue transportar itens com um peso máximo de *P*, independentemente da capacidade total da mochila. Que alteração se teria que introduzir para comportar esta restrição adicional?

4.
Como se poderia adaptar o algoritmo para o caso em que existem duas mochilas com a mesma capacidade?

5.
Considere a implementação do algoritmo de busca em profundidade (*depth-first-search*) no Anexo B (corresponde ao código no Cap. 12 do manual). Complete a documentação.

6.
Descreva um algoritmo de busca num grafo que em vez de ser em profundidade é em largura (*breadth-first-search*).

7.
Considere o seguinte grafo direcionado:



(a) Encontre o caminho mais curto entre A e G usando busca em profundidade. Apresente os vários passos do algoritmo.

(b) Encontre o caminho mais curto entre A e G usando busca em largura (*breadth-first-search)*. Apresente os vários passos do algoritmo.

(c) Repita o exercício para encontrar o caminho entre C e D.

8.

Considere o seguinte excerto do mapa das auto-estradas Portuguesas, com as seguintes cidades e distâncias entre elas:

A1: Lisboa <40km> Carregado <42km> Santarém
A8: Caldas da Rainha <65km> Malveira <17km> Loures <8km> Lisboa
A15: Caldas da Rainha <46km> Santarém
A10/A9: Carregado <33km> Loures <21km> Oeiras

Pretende-se encontrar o caminho mais curto entre duas dadas cidades, usando uma abordagem baseada em grafos.

(a) Construa o grafo respectivo.

(b) Descreva um algoritmo de busca em profundidade que encontre o caminho no grafo entre duas cidades com menor peso (menor distância total neste caso).

(c) Use esse algoritmo para encontrar o caminho mais curto entre a Malveira e Santarém.

Problemas

1.
Considere a concretização em Python do algoritmo ótimo, ou força bruta, para a resolução do problema da mochila 0/1 apresentado abaixo no Anexo A (corresponde ao código no Cap. 12 do manual). Ajuste o código de forma a que este entregue a lista de todas as soluções ótimas que existirem (e não apenas uma).

2.
Considere a concretização em Python do algoritmo guloso (greedy) para a resolução do problema da mochila 0/1 apresentado abaixo no anexo A (corresponde ao código no Cap. 12 do manual).

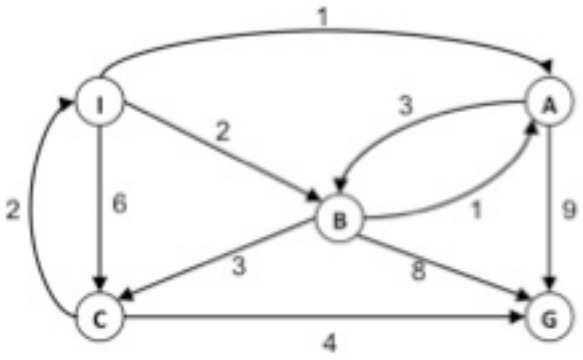
(a) Suponha que em vez de uma mochila, havia duas mochilas para encher. Discuta as alternativas para adaptar o algoritmo guloso. Ajuste o código para que este acomode duas dessas alternativas. Leve a efeito experiências que permitam estudar os prós e contras destas opções.

(b) Suponha o seguinte cenário: cada item na coleção de partida tem um valor e um prazo de validade. Agora há dois objetivos: encher a mochila de forma a otimizar o valor total e a duração dos itens que nela são carregados. Discuta alternativas para adaptar o algoritmo guloso. Ajuste o código para que este acomode duas dessas alternativas. Leve a efeito experiências que permitam estudar os prós e contras destas opções.

(c) Suponha o seguinte cenário: cada item na coleção de partida tem um valor e um prazo de validade. Agora há duas condições: a mochila só pode ser carregada até um limite de peso e só pode acomodar até um número máximo de itens. Discuta alternativas para adaptar o algoritmo guloso. Ajuste o código para que este acomode duas dessas alternativas. Leve a efeito experiências que permitam estudar os prós e contras destas opções.

3.
Considere a concretização em Python do algoritmo de busca em profundidade do caminho mais curto (depth-first search shortest-path algorithm) apresentado abaixo no Anexo B (corresponde ao código no Cap. 12 do manual). Por ajustamento deste programa, transforme-o para que concretize o algoritmo de busca em profundidade num grafo pesado do caminho que minimiza a soma dos pesos dos respetivos arcos.

4.
Considere of grafo seguinte, que representa um conjunto de lojas na baixa de uma cidade e as ruas (de sentido único) que as ligam, com as respetivas distâncias em quilómetros. Aplique o programa obtido no problema anterior para determinar o caminho mais curto entre I e G.


1
I
A
2
3
2
6
B
1
9
3
8
C
G
4

Anexo A

```
class Item(object):
    """
    Item
    """

    def __init__(self, n, v, w):
        """
        Constructor

        Requires:
        n string, represents name;
        v float, represents value;
        w float, represents weight.
        Ensures:
        object of type Item created with given paramneters.
        """
        self._name = n
        self._value = float(v)
        self._weight = float(w)

    def getName(self):
        """
        Name of item

        Ensures:
        Name of item
        """
        return self._name

    def getValue(self):
        """
        Value of item

        Ensures:
        Value of item
        """
        return self._value

    def getWeight(self):
        """
        Weight of item

        Ensures:
        Weight of item
        """
        return self._weight
```

```
    def __str__(self):
        """
        String representation of Item

        Ensures:
        String representation of Item in formta
        < name, valie, weight>
        """
        result = '<' + self._name + ', ' + str(self._value)\
                 + ', ' + str(self._weight) + '>'
        return result


def value(item):
    """
    Value of a given item

    Requires:
    item Item
    Ensures:
    Value of given item
    """
    return item.getValue()


def weightInverse(item):
    """
    Inverse of weight of a given item

    Requires:
    item Item
    Ensures:
    Inverse of weight of given item
    """
    return 1.0/item.getWeight()


def density(item):
    """
    Density of a given item

    Requires:
    item Item
    Ensures:
    Density of given item
    """
    return item.getValue()/item.getWeight()


def buildItems():
    """
    List of items

    Ensures:
    List with items
```

```
    <clock, 175, 10>
    <painitng, 90, 9>
    <radio, 20, 4>
    <vase, 50, 2>
    <book, 10, 1>
    <computer, 200, 20>
    """
    names = ['clock', 'painting', 'radio', 'vase', 'book', 'computer']
    values = [175, 90, 20, 50, 10, 200]
    weights = [10, 9, 4, 2, 1, 20]
    Items = []
    for i in range(len(values)):
        Items.append(Item(names[i], values[i], weights[i]))
    return Items


## greedy


def greedy(items, maxWeight, keyFunction):
    """
    Knapsack filling for given items, constraint and
    auxiliary function - solving with greedy algorithm

    Requires:
    items list of Items;
    maxWeight float, represents contraint of greedy
    algorithm, i.e. capacity of knapsack in terms of maximum weight;
    keyFunction function, reprsenting auxiliary function
    for selection.
    Ensures:
    Knapsack filled with maximum possible agregated value
    of items up to the maxWeight capacity of knapsack
    along keyFunction auxiliary function, according
    to greedy algorithm (local maximum).
    """
    itemsCopy = sorted(items, key=keyFunction, reverse = True)
    result = []
    totalValue = 0.0
    totalWeight = 0.0
    for i in range(len(itemsCopy)):
        if (totalWeight + itemsCopy[i].getWeight()) <= maxWeight:
            result.append(itemsCopy[i])
            totalWeight += itemsCopy[i].getWeight()
            totalValue += itemsCopy[i].getValue()
    return (result, totalValue)
```

```
def testGreedy(items, constraint, keyFunction):
    """
    Testing greedy algorithm to solve knapsack filling

    Requires:
    items list of Item;
    constraint float, represent capacity of knapsack
    in terms of maximum weight;
    keyFunction function, auxiliary function for selection
    Ensures:
    Print to stdout knapsack filled along greedy fucntion.
    """
    taken, val = greedy(items, constraint, keyFunction)
    print('Total value of items taken = ', val)
    for item in taken:
        print(' ', item)


def testGreedys(maxWeight = 20):
    """
    Testing greedy algorithm to solve knapsack filling
    with 20 maximum weight, and with 3 auxilary functions
    for selection.

    Requires:
    maxWeight float, representing capacity of knapsack
    in terms of maximum capacity
    Ensures:
    Print to stdout knapsack filled along greedy fucntion
    with each auxiliary function value, weightInverse and
    density
    """
    items = buildItems()
    print('Use greedy by value to fill knapsack of size', maxWeight)
    testGreedy(items, maxWeight, value)
    print('\nUse greedy by weight to fill knapsack of size',
maxWeight)
    testGreedy(items, maxWeight, weightInverse)
    print('\nUse greedy by density to fill knapsack of size',
maxWeight)
    testGreedy(items, maxWeight, density)

testGreedys()
```

```
## brute force


def getBinaryRep(n, numDigits):
    """
    Binary representation of a given decimal integer

    Requires:
    n int >= 0
    numDigits int, representing the nyumber of digits
    in binary representation
    Ensures:
    String of length numDigits that is the binary
    representation of n
    """
    result = ''
    while n > 0:
        result = str(n%2) + result
        n = n//2
    if len(result) > numDigits:
        raise ValueError('not enough digits')
    for i in range(numDigits - len(result)):
        result = '0' + result
    return result


def genPowerset(L):
    """
    Power set of a given set

    Requires:
    L list
    Ensures:
    List Power(L)
    """
    powerset = []
    for i in range(0, 2**len(L)):
        binStr = getBinaryRep(i, len(L))
        subset = []
        for j in  range(len(L)):
            if binStr[j] == '1':
                subset.append(L[j])
        powerset.append(subset)
    return powerset


def chooseBest(pset, maxWeight, getVal, getWeight):
    """
    Knapsack filling for given items and constraint - solving
    with brute force algorithm

    Requires:
    pset list, with power list of list of Items;
    maxWeight float, represents contraint of chooseBest
```

```
    algorithm, i.e. capacity of knapsack in terms of maximum weight;
    getVal function getVal;
    getWeight function getWeight.
    Return:
    Knapsack filled with maximum possible agregated value
    of items up to the maxWeight capacity of knapsack
    along keyFunction auxiliary function, according
    to brute force algorithm (global maximum).
    """
    bestVal = 0.0
    bestSet = None
    for items in pset:
        itemsVal = 0.0
        itemsWeight = 0.0
        for item in items:
            itemsVal += getVal(item)
            itemsWeight += getWeight(item)
        if itemsWeight <= maxWeight and itemsVal > bestVal:
            bestVal = itemsVal
            bestSet = items
    return (bestSet, bestVal)


def testBest(maxWeight = 20):
    """
    Testing brute force algorithm to solve knapsack filling
    with 20 maximum weight.

    Requires:
    maxWeight float, representing capacity of knapsack
    in terms of maximum capacity
    Ensures:
    Print to stdout knapsack filled along chooseBest function.
    """
    items = buildItems()
    pset = genPowerset(items)
    taken, val = chooseBest(pset, maxWeight, Item.getValue,
Item.getWeight)
    print('\nUse optimal solution (brute force) to fill knapsack of
size', maxWeight)
    print('Total value of items taken =', val)
    for item in taken:
        print(item)


testBest()
```

Anexo B

```
class Node(object):
    def __init__(self, name):
        """
        Requires: name is a string
        """
        self._name = name
    def getName(self):
        return self._name
    def __str__(self):
        return self._name



class Edge(object):
    def __init__(self, src, dest):
        """
        Requires: src and dst Nodes
        """
        self._src = src
        self._dest = dest
    def getSource(self):
        return self._src
    def getDestination(self):
        return self._dest
    def __str__(self):
        return self._src.getName() + '->' + self._dest.getName()



class Digraph(object):
    #nodes is a list of the nodes in the graph
    #edges is a dict mapping each node to a list of its children
    def __init__(self):
        self._nodes = []
        self._edges = {}
    def addNode(self, node):
        if node in self._nodes:
            raise ValueError('Duplicate node')
        else:
            self._nodes.append(node)
            self._edges[node] = []
    def addEdge(self, edge):
        src = edge.getSource()
        dest = edge.getDestination()
        if not(src in self._nodes and dest in self._nodes):
            raise ValueError('Node not in graph')
        self.edges[src].append(dest)
    def childrenOf(self, node):
        return self.edges[node]
    def hasNode(self, node):
        return node in self._nodes
```

```
    def __str__(self):
        result = ''
        for src in self._nodes:
            for dest in self._edges[src]:
                result = result + src.getName() + '->'\
                + dest.getName() + '\n'
        return result


class Graph(Digraph):
    def addEdge(self, edge):
        Digraph.addEdge(self, edge)
        rev = Edge(edge.getDestination(), edge.getSource())
        Digraph.addEdge(self, rev)


def printPath(path):
    """
    Requires: path a list of nodes
    """
    result = ''
    for i in range(len(path)):
        result = result + str(path[i])
        if i != len(path) - 1:
            result = result + '->'
    return result


def DFS(graph, start, end, path, shortest):
    """
    Requires:
    graph a Digraph;
    start and end nodes;
    path and shortest lists of nodes
    Ensures:
    a shortest path from path to end in graph
    """
    path = path + [start]
    print('Current DFS path:', printPath(path))
    if start == end:
        return path
    for node in graph.childrenOf(start):
        if node not in path: #avoid cycles
            if shortest == None or len(path) < len(shortest):
                newPath = DFS(graph, node, end, path, shortest)
                if newPath != None:
                    shortest = newPath
    return shortest
```

```
def search(graph, start, end):
    """
    Requires:
    graph  a Digraph;
    start and end are nodes
    Ensures:
    shortest path from start to end in graph
    """
    return DFS(graph, start, end, [], None)


def testSP():
    nodes = []
    for name in range(6): #Create 6 nodes
        nodes.append(Node(str(name)))
    g = Digraph()
    for n in nodes:
        g.addNode(n)
    g.addEdge(Edge(nodes[0],nodes[1]))
    g.addEdge(Edge(nodes[1],nodes[2]))
    g.addEdge(Edge(nodes[2],nodes[3]))
    g.addEdge(Edge(nodes[2],nodes[4]))
    g.addEdge(Edge(nodes[3],nodes[4]))
    g.addEdge(Edge(nodes[3],nodes[5]))
    g.addEdge(Edge(nodes[0],nodes[2]))
    g.addEdge(Edge(nodes[1],nodes[0]))
    g.addEdge(Edge(nodes[3],nodes[1]))
    g.addEdge(Edge(nodes[4],nodes[0]))
    sp = search(g, nodes[0], nodes[5])
    print('Shortest path found by DFS:', printPath(sp))


testSP()
```