APA LAB 3

Cojocaru Andreea - Cristina

December 4, 2016

1 Tasks

1. Study of Greedy mehtodes
2. Python implementation of Kruskal and Prim algorithms
3. Impiric and comparatic analysis of Kruskal and Prim algorithms

2 Study of Greedy mehtodes

A greedy algorithm is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum. In many problems, a greedy strategy does not in general produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a global optimal solution in a reasonable time.

2.1 Counting Coins

This problem is to count to a desired value by choosing the least possible coins and the greedy approach forces the algorithm to pick the largest possible coin. If we are provided coins of €1, 2, 5 and 10 and we are asked to count 18 then the greedy procedure will be:

1. Select one €10 coin, the remaining count is 8
2. Then select one €5 coin, the remaining count is 3
3. Then select one €2 coin, the remaining count is 1
4. And the selection of one €1 coins solves the problem

Though, it seems to be working fine, for this count we need to pick only 4 coins. But if we slightly change the problem then the same approach may not be able to produce the same optimum result.

For the currency system, where we have coins of 1, 7, 10 value, counting

coins for value 18 will be absolutely optimum but for count like 15, it may use more coins than necessary. For example, the greedy approach will use 10 + 1 + 1 + 1 + 1 + 1, total 6 coins. Whereas the same problem could be solved by using only 3 coins (7 + 7 + 1)

Hence, we may conclude that the greedy approach picks an immediate optimized solution and may fail where global optimization is a major concern.

2.2 Examples

Most networking algorithms use the greedy approach. Here is a list of few of them:

1. Travelling Salesman Problem
2. Prim's Minimal Spanning Tree Algorithm
3. Kruskal's Minimal Spanning Tree Algorithm
4. Dijkstra's Minimal Spanning Tree Algorithm
5. Knapsack Problem

3 Python implementation of Kruskal and Prim algorithms

3.1 Python implementation of Kruskal algorithm

```
from operator import itemgetter

class DisjointSet(dict):
    def add(self, item):
        self[item] = item

    def find(self, item):
        parent = self[item]

        while self[parent] != parent:
            parent = self[parent]

        self[item] = parent
        return parent

    def union(self, item1, item2):
        self[item2] = self[item1]

def kruskal( nodes, edges ):
    forest = DisjointSet()
```

```
    mst = []
    for n in nodes:
        forest.add( n )

    sz = len(nodes) - 1
    for e in sorted( edges, key=itemgetter( 2 ) ):
        n1, n2, _ = e
        t1 = forest.find(n1)
        t2 = forest.find(n2)
        if t1 != t2:
            mst.append(e)
            cost += _
            sz -= 1
            if sz == 0:
                return mst
            forest.union(t1, t2)
```

3.2 Python implementation of Prim algorithm

```
from collections import defaultdict
from heapq import *

def prim( nodes, edges ):
    conn = defaultdict( list )
    for n1,n2,c in edges:
        conn[ n1 ].append( (c, n1, n2) )
        conn[ n2 ].append( (c, n2, n1) )

    mst = []
    used = set( [nodes[ 0 ]] )
    usable_edges = conn[ nodes[0] ][:]
    heapify( usable_edges )

    while usable_edges:
        cost, n1, n2 = heappop( usable_edges )
        if n2 not in used:
            used.add( n2 )
            mst.append( ( n1, n2, cost ) )

            for e in conn[ n2 ]:
                if e[ 2 ] not in used:
                    heappush( usable_edges, e )
    return mst
```

3.3 Results

To test the algorithms I used the following data:

```
 nodes = list( "123456789" )
edges = [   ("1", "5",1),
            ("1", "3",1),
            ("5", "8",1),
            ("9", "7",1),
            ("5", "6",2),
            ("1", "2",2),
            ("2", "5",2),
            ("5", "7",2),
            ("5", "6",2),
            ("2", "4",3),
            ("5", "9",3),
            ("6", "8",3),
            ("8", "9",4),
            ("4", "7",4)
        ]
```

The results are the following:

```
 ([('1', '5', 1),
   ('1', '3', 1),
   ('5', '8', 1),
   ('9', '7', 1),
   ('5', '6', 2),
   ('1', '2', 2),
   ('5', '7', 2),
   ('2', '4', 3)])
Kruskal time result: 0:00:00
[('1', '3', 1),
   ('1', '5', 1),
   ('5', '8', 1),
   ('1', '2', 2),
   ('5', '6', 2),
   ('5', '7', 2),
   ('7', '9', 1),
   ('2', '4', 3)]
Prim time result: 0:00:00
```

4 Impiric and comparatic analysis of Kruskal and Prim algorithms

4.1 Time complexity

After rulling my test I discovered that no matter hoe complex my graph gets the running time is always 0:00:00 that is because my graphs were not complex enough for my computer, therefore it is petter to make an impiric analysis.

The time complexity of Prim's algorithm depends on the data structures used for the graph and for ordering the edges by weight, which can be done using a priority queue.

A simple implementation of Prim's, using an adjacency matrix or an adjacency list graph representation and linearly searching an array of weights to find the minimum weight edge, to add requires O($|V|^2$) running time. However, this running time can be greatly improved further by using heaps to implement finding minimum weight edges in the algorithm's inner loop.

A first improved version uses a heap to store all edges of the input graph, ordered by their weight. This leads to an O($|E|$ log $|E|$) worst-case running time. But storing vertices instead of edges can improve it still further. The heap should order the vertices by the smallest edge-weight that connects them to any vertex in the partially constructed minimum spanning tree (MST) (or infinity if no such edge exists). Every time a vertex v is chosen and added to the MST, a decrease-key operation is performed on all vertices w outside the partial MST such that v is connected to w, setting the key to the minimum of its previous value and the edge cost of (v,w).

Where E is the number of edges in the graph and V is the number of vertices, Kruskal's algorithm can be shown to run in O(E log E) time, or equivalently, O(E log V) time, all with simple data structures. These running times are equivalent because:

1. E is at most V^2 and log V^2 = 2 log V is O(log V)
2. Each isolated vertex is a separate component of the minimum spanning forest. If we ignore isolated vertices we obtain V $\leq$ 2E, so log V is O(log E).

5 Conclusion

In this laboratory work I revised my knowledghe of the Greedy Algorithms and I implemented two of the meny existing: Prim's Minimal Spanning Tree Algorithm and Kruskal's Minimal Spanning Tree Algorithm. I implemented both algorithms in python and made three tests which were in my oppinion relatively complex, sadly they were not complex enough for my computer to spend more than 1 milisecond. Therefore, I decided to make an impiric analysis and I concluded that I should use Prim's algorithm when I have a graph with lots of adges. Prim's algorithm is significantly faster in the limit when I've got

a really dense graph with many more edges than vertices. Kruskal performs better in typical situations (sparse graphs) because it uses simpler data structures.Kruskal's algorithm will grow a solution from the cheapest edge by adding the next cheapest edge, provided that it doesn't create a cycle. Prim's algorithm will grow a solution from a random vertex by adding the next cheapest vertex, the vertex that is not currently in the solution but connected to it by the cheapest edge. If you implement both Kruskal and Prim, in their optimal form: with a union find and a finbonacci heap respectively, then you will note how Kruskal is easy to implement compared to Prim. Prim is harder with a fibonacci heap mainly because you have to maintain a book-keeping table to record the bi-directional link between graph nodes and heap nodes. With a Union Find, it's the opposite, the structure is simple and can even produce directly the mst at almost no additional cost.

Doing this laboratory work I concluded that before proceding to solve any problem using any algorithm I must analyse my input information fary carefully, and then I should analyse the complexity of my algorithms and only after that i can implement one. This way I will maximize my efficiency, I will not spend time on implementing an algorithm that will not work with my set of data and I will not waste my resouces on implementing an algorithm knowing that there is another that is much much faster.