SHELDONAIR

ALGORITHMS PROJECT 5

Anqi Chen, Francesco Radealli, Fangqing Yuan
LUISS GUIDO CARLI

- **Original Problem**: *Dr. Cooper would like to know the minimum number of legs required to fly from one city to another city, given the map of SheldonAir flights map.*
- **Problem behind:** *Given two vertices u and v of an undirected, unweighted graph (with n vertices and m edges), compute the length of a shortest path between u and v.*
- *In our python3 codes, a graph is a dictionary which keys are vertices and values are list of vertices connected to corresponding key.*
- **Language:** *Python 3*
- **Libraries:** *math, random, time, dequeue, numpy, matplotlib*
- **Test Environment:** *PC*

Naive Approach

Floyd & Warshall (All Pairs Shortest Paths)

- Application of *dynamic programming*
- **Idea:** Starting from the *adjacency matrix(1 means an edge, math.inf means no edge)*, build a three-dimensional minimum distance matrix
- Can reduce space to $O(n^2)$ (we use n^3 in our codes)

Time Complexity: $O(n^3)$

Space Complexity: $O(n^2)$ – *n x n array (matrix)*

Bread First Search

Can we do better than APSP?

Since the graph in unweighted, actually *Bread First Search* provides a solution to the problem!

Time Complexity: $O(n + m)$

Space Complexity: $O(n + m)$ – *The graph itself*

Python Codes – Algorithms – User Guide

- **BFS_Shortest_Path:** *implementation of BFS **(Solution1)***
  - ✓ *Insert your own sample graph implemented as an adjacency matrix*
  - ✓ *Call the «short» function, passing as parameters the graph and the two nodes*

```python
SampleGraph = {"A": ["B", "D", "C"],
               "B": ["C", "A"],
               "C": ["A", "D", "B"],
               "D": ["C", "A"]}

print(short(SampleGraph, "B", "D"))
```

- **DP_Shortest_Path:** *implementation of APSP **(Solution2)***
  - ✓ *Insert your own sample graph implemented as an adjacency matrix*
  - ✓ *Call the «cube» function, passing as parameter the graph*
  - ✓ *Call the «nodes» function, passing as parameter the graph*
  - ✓ *Call the «short» function, passing as parameters the graph and the two nodes*

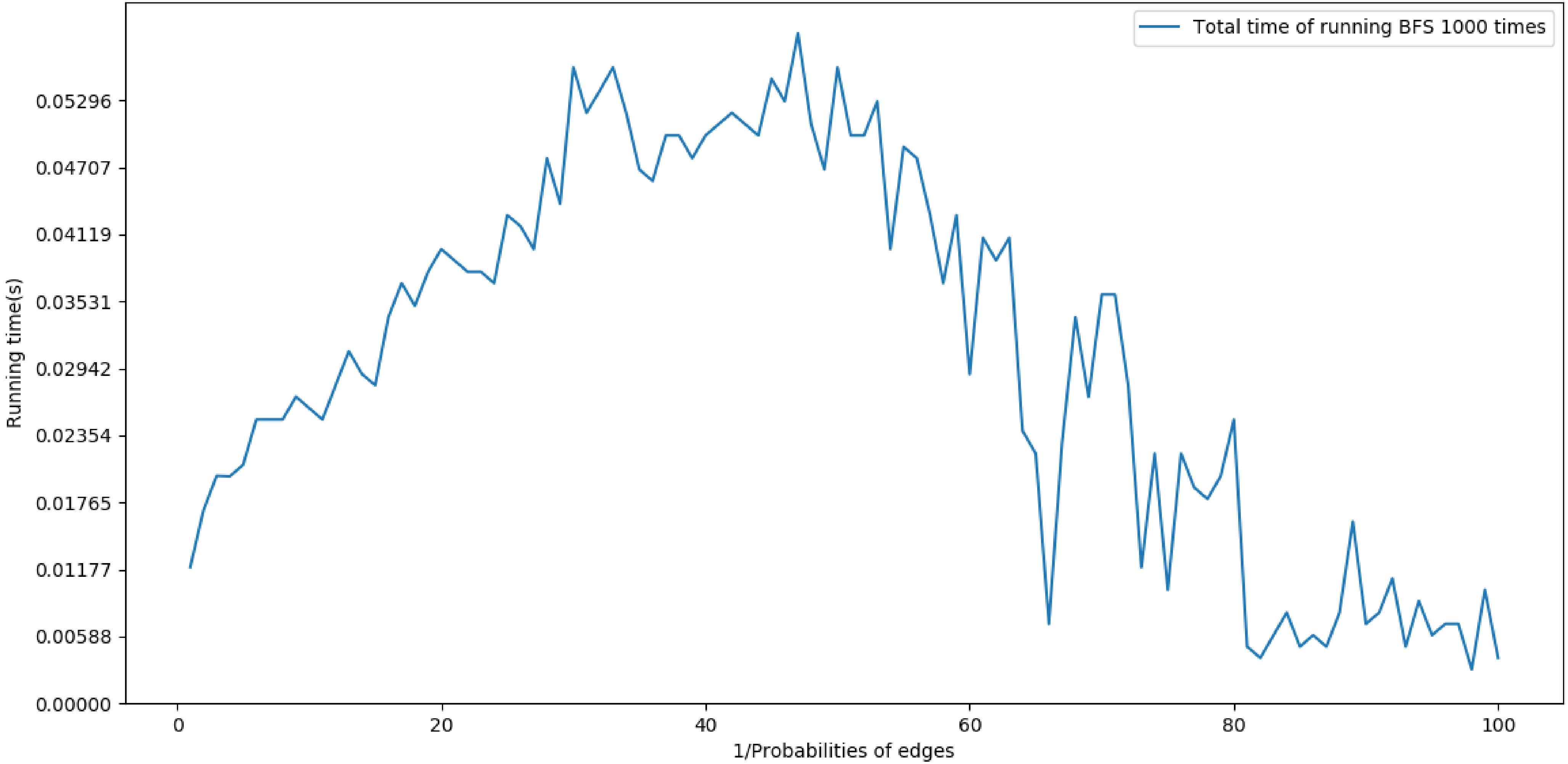
```python
SampleGraph = {"A": ["B", "D", "C"],
               "B": ["C", "A"],
               "C": ["A", "D", "B"],
               "D": ["C", "A"]}
M = cube(SampleGraph)
vertices = nodes(SampleGraph)
print(short(SampleGraph, "B", "D"))
```

Other Python Codes - Tests

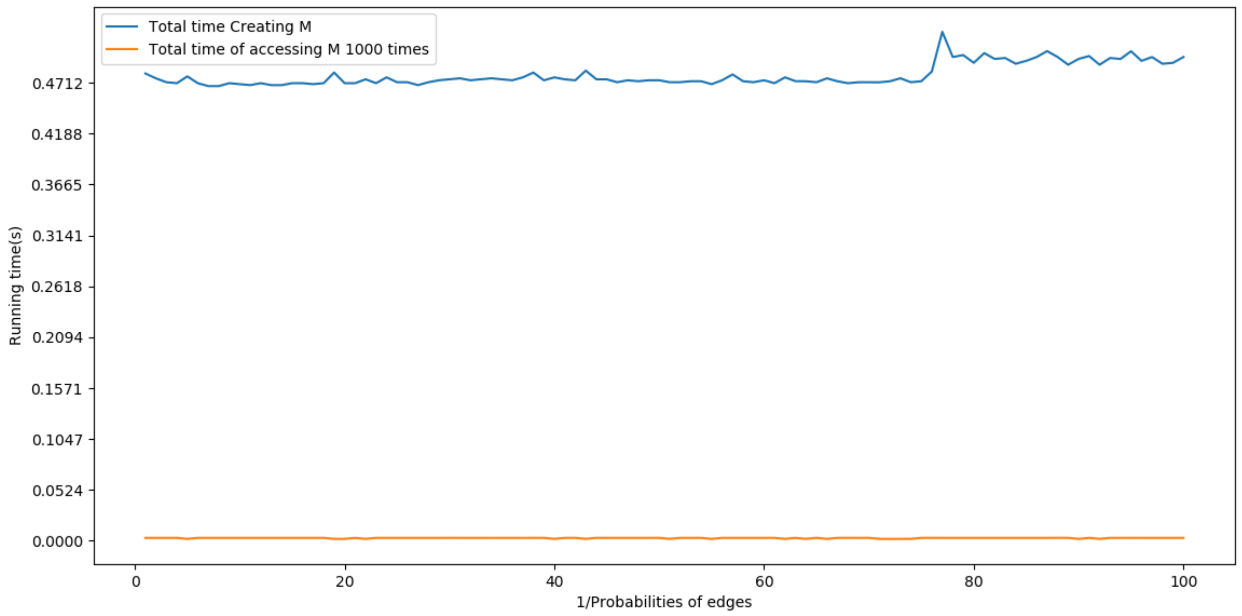
- **test_BFS:** *implementation of BFS. Test on inputs with different probability of having an edge between two nodes.*
- **test_DP1:** *implementation of APSP. Test on inputs with different probability of having an edge between two nodes.*
- **test_DP2:** *implementation of APSP. Test on input graphs with different number of nodes.*
- **test_stress:** *implementation of a Stress Test to check correcteness.*
- **graph_random:** *to generate a random graph.*

Algorithm Implementation: Plots - BFS

Graph info: n=100

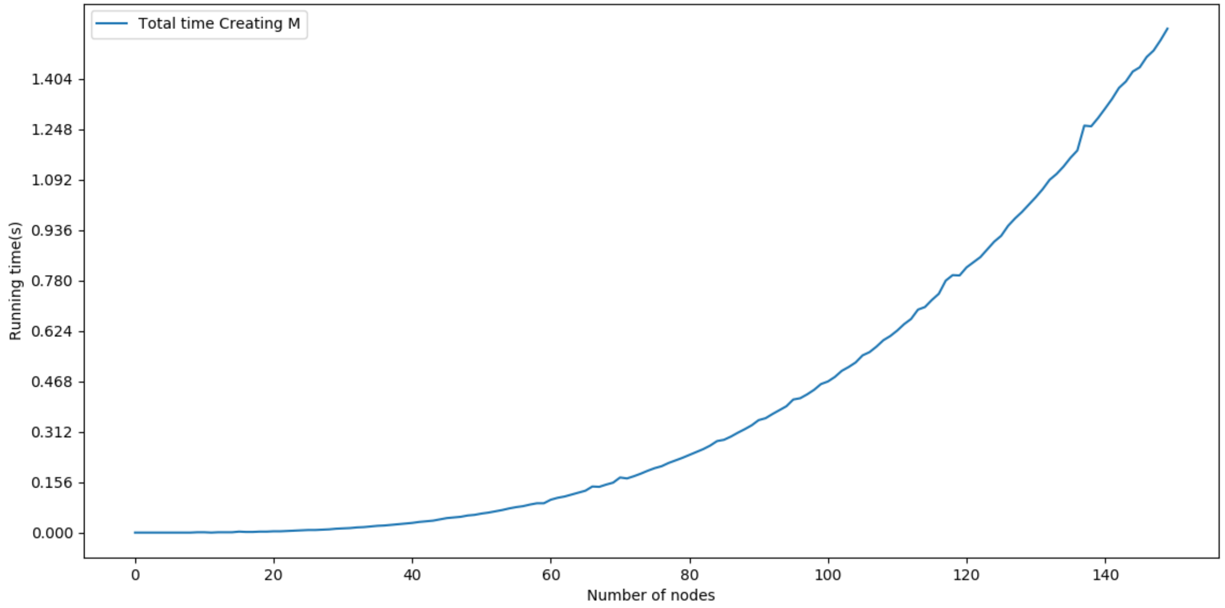


Algorithm Implementation: Plots – DP1&DP2



n=100, time of creating matrix(blue line), and time of accessing(orange line, almost 0).

n=1to150, time of building distance matrix. The plot is a perfect n^3 graph.



ANALYSIS OF PLOTS

- Why BFS test plot has a peak around 40?
  - The x axis represents the reciprocal of probability of having an edge between two vertices. In both high and low probabilities, BFS can perform very fast, while a middle amount of probability slow it down.
- Why DP cost almost same time whatever the probability is?
  - DP is always creating the distance matrix in O(n^3) , so m doesn't matter the total time.

Which algorithm is better?

From our Time Complexity Analysis, BFS turns out to be faster - O(n + m)

Actually, practical implementation matters!

In the Dynamic Programming approach, the demanding step in terms of computational resources is ***building the matrix***, while accessing it is extremely fast!

An Amortized Analysis for a huge number of accesses might prove better bounds

In practice, SheldonAir could follow the DP approach by:

- Building the table just once and storing it (Time $O(n^3)$ needed)
- All the future accesses will then be possible in constant time (*Close to 0, as shown in the Plot!*)
- *Or just uses BFS if he wants only one pair of u and v.*
- *You're welcome, Sheldon!*