CS301

2022-2023 Spring

Project Report

Group 126

Kerim Demir (28853)

Steven El Khaldi (30048)

1. Problem Description

Describe your problem both as intuitively and formally. If possible, talk about the applications (where this problem) might be used in practice.

State the hardness of your problem in the form of a theorem. For example, give a theorem claiming that your problem is NP-complete, or NP-hard.

For the proof of this theorem, you can simply cite (refer to) an appropriate source in the literature which gives this proof, or you can give an explicit proof in your report. In the case you give an explicit proof, and this proof is not a novel proof suggested by your group (which is ok for your report), you must still give the citation to the original paper/book from which you got this proof.

2. Algorithm Description

a. Brute Force Algorithm

Find a correct/exact/brute force algorithm for your problem and explain the algorithm in detail (Exponential/Factorial Time, Not Efficient). If you want, you can also design an algorithm yourself. Also give a pseudo-code of the algorithm. If the algorithm follows an algorithm design technique, e.g., divide-and-conquer, dynamic programming, etc., you need to mention this and explain why you think the algorithm is using this technique. (Guaranteed solution no matter the computational complexity.)

b. Heuristic Algorithm

A polynomial time approximation algorithm for the vertex cover problem is a known **greedy** one, the Approx-Vertex-Cover algorithm, that performs the following steps:

- Initialize an empty set for the vertex cover.
- Pick a random edge in the graph, and add both its vertices to the empty vertex cover set. Then, mark the edge as covered, and for each edge incident on either of the two vertices of the edge, also mark it as covered.
- Repeat the process until all edges are covered.

A pseudo-code of the algorithm could be the following:

```
Set vertexCover to empty set
Set uncoveredEdges to all edges E
While uncoveredEdges is not empty:
   Set currentEdge to a random edge in uncoveredEdges
   Add both vertices u and v of currentEdge to vertexCover
   Remove currentEdge and all other edges in uncoveredEdges having u or v as one of its
vertices from uncoveredEdges
Return vertexCover
```

As previously mentioned, this algorithm follows the *greedy* algorithm design technique, because instead of considering the problem as a whole, it considers uncovered edges on their own and all other edges incident on both vertices, solving the problem locally instead of globally.

Ratio bound

Statement: The ratio bound for the aforementioned greedy algorithm for the vertex cover problem has a ratio bound of 2.

Proof: Consider the set A of all edges randomly chosen as currentEdge in the pseudo-code provided above. Since all edges incident on either vertex of currentEdge are also removed from uncoveredEdges during each iteration, any next randomly chosen currentEdge can't possibly have either of the previous currentEdge's two vertices. Hence, no two edges in the set A have a common vertex, resulting in the deduction that:

*The length | vertexCover | of the returned vertex cover, which only includes both endpoints of each edge in the set A, is equal to 2 * | A |.*

However, an optimal vertex cover would have to include at least one of the two vertices of each edge in A in order to cover the set A. Hence, we can also deduce that:

The length | optimalVertexCover | of an optimal vertex cover would be >= | A |.

Therefore, we can furthermore deduce that:

$$| optimalVertexCover | >= | greedyVertexCover | / 2$$

$$\mathbf{2 >= | greedyVertexCover | / | optimalVertexCover |}$$[1]

[1] https://www.cl.cam.ac.uk/teaching/1415/AdvAlgo/lec8_ann.pdf

3. Algorithm Analysis

a. Brute Force Algorithm

- *Claim and show that the algorithm works correctly, possibly in the form of a theorem*
- *For the complexity analysis, drive the worst-case time complexity. Try not to give upper bounds which are too loose. If possible, try to give a tight upper bound by using* $\boldsymbol{\Theta}$.
- *Optionally, you can also consider the complexity of the algorithm for resources other than time, e.g., the space complexity.*

b. Heuristic Algorithm

Claim: The Approx-Vertex-Cover greedy algorithm always computes a vertex cover whose length is at most double the length of the minimum vertex cover of the given graph.

Proof: For the claim to be correct, two things must be proven: (a) that the algorithm always produces a correct vertex cover, and (b) that the produced vertex cover's length is at most double the length of the optimal (minimum) vertex cover of the graph.

(a) During the algorithm, we iterate over all uncovered edges (initially the set of all edges E). At each iteration, we randomly choose an uncovered edge (u, v), include both endpoints in the vertex cover (initially empty), and remove all edges having at least one of u and v as its endpoint from the set of uncovered edges. We keep iterating until there are no more uncovered edges, indicating that all edges E in the graph have at least one endpoint in the produced vertex cover, making it a correct vertex cover.

(b) As proven in **2-b**, the Approx-Vertex-Cover algorithm has a ratio bound of 2, meaning that the length of the vertex cover it produces is less than or equal to double the length of the optimal vertex cover.

Complexity analysis: The worst-case time complexity of the algorithm can be found to be $\boldsymbol{\Theta(|E|)}$, where |E| is the number of edges in the graph, through the following analysis:

- Creating the initial empty vertex cover set takes constant $\boldsymbol{\Theta}(1)$ time.
- Copying the set E to the initial uncoveredEdges set takes $\boldsymbol{\Theta}(|E|)$ time.
- When iterating over uncoveredEdges until it is empty, we see each edge exactly once. This is because for each uncovered edge, we remove all edges incident on either of its endpoints from the uncoveredEdges set, so each edge is either iterated over as an uncovered edge and then removed from the set or removed from the set during the iteration of another uncovered edge for being incident on either of its endpoints. Therefore, the time complexity of this while loop is $\boldsymbol{\Theta}(|E|)$. Adding the endpoints to the vertex cover takes constant time and we do it for a number of edges less than |E|, so the complexity stays $\boldsymbol{\Theta}(|E|)$.
- Hence, the total worst-case time complexity of the algorithm is $\boldsymbol{\Theta}(1 + |E| + |E|) = \boldsymbol{\Theta(|E|)}$.

- *Optionally, you can also consider the complexity of the algorithm for resources other than time, e.g., the space complexity.*

4. Sample Generation (Random Instance Generator)

- *Implement/find a parametric (in terms of the size of the problem) algorithm to produce random sample inputs for your problem.*
- *Put pseudo codes and explanation of the algorithm to your reports.*

5. Algorithm Implementations

a. Brute Force Algorithm

Implement the brute force algorithm and perform an initial testing of the implementation by using 15-20 samples using the sample generator tool of Section 4. Note that, although you can implement this algorithm yourself (if you want to), it is also fine if you use a code that you find from the internet. However, you should be able to install and run it. Also, you need to get familiar with the source code to be able to answer any questions about the code.

Report the results of the initial testing by giving the number and the size of the instances tried. Report any failures and related fixes.

b. Heuristic Algorithm

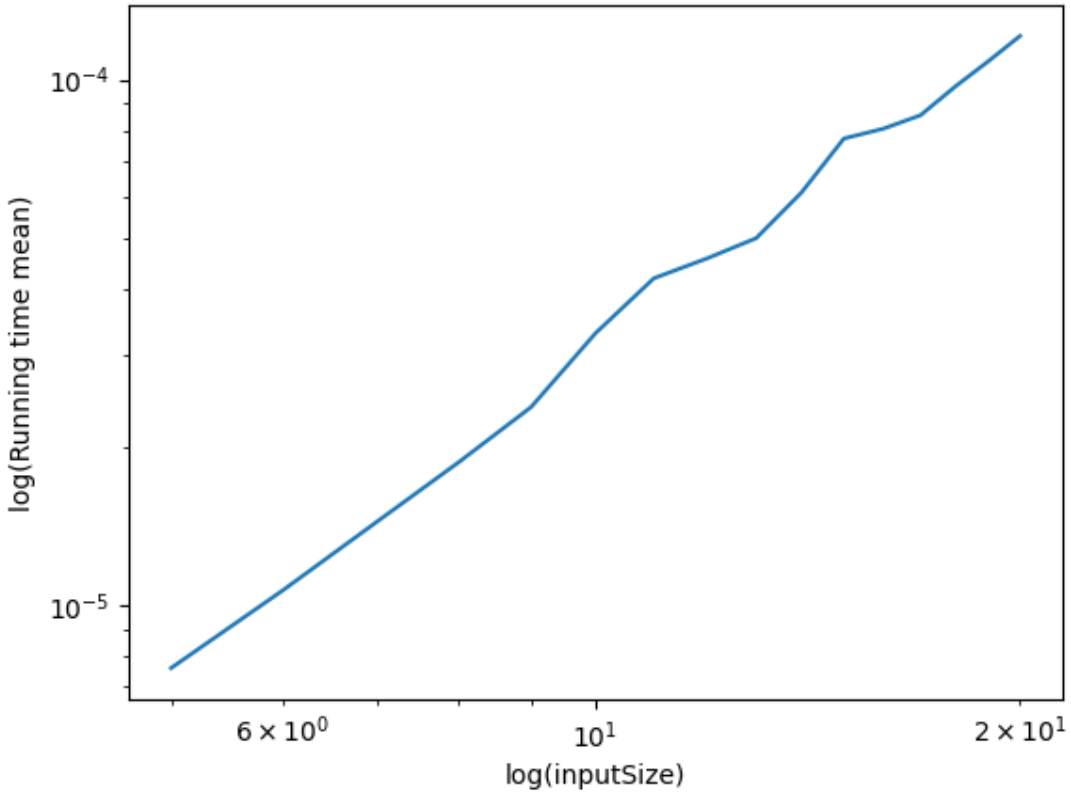
The Algorithm:

```
def remove(edges, u, v):
    newEdges = []
    for edge in edges:
        if (u not in edge) and (v not in edge):
            newEdges.append(edge)
    return newEdges


def vertex_cover_heuristic(edges, n_vertices, k):
     if k < 0 or k > n_vertices or n_vertices == 0 or edges
== []:
        return [False, []]
    vertexCover = []
    UC = edges
    while len(UC) > 0:
        CE = UC[random.randint(0, len(UC)-1)]
        vertexCover.append(CE[0])
        vertexCover.append(CE[1])
        UC = remove(UC, CE[0], CE[1])

    if len(vertexCover) <= k:
        return [True, vertexCover]
    else:
        return [False, vertexCover]
```

6. Experimental Analysis of The Performance (Performance Testing)

With 90% Confidence Level and N value as 100 000, t is 1.645. When we calculate the confidence interval [a-b, a+b] we get a (b/a) value that is smaller than 0.1 as requested of us.

After the calculations we get a logarithmic inputSize-Running Time graph, so when we put our values in a loglog chart we get the image below:

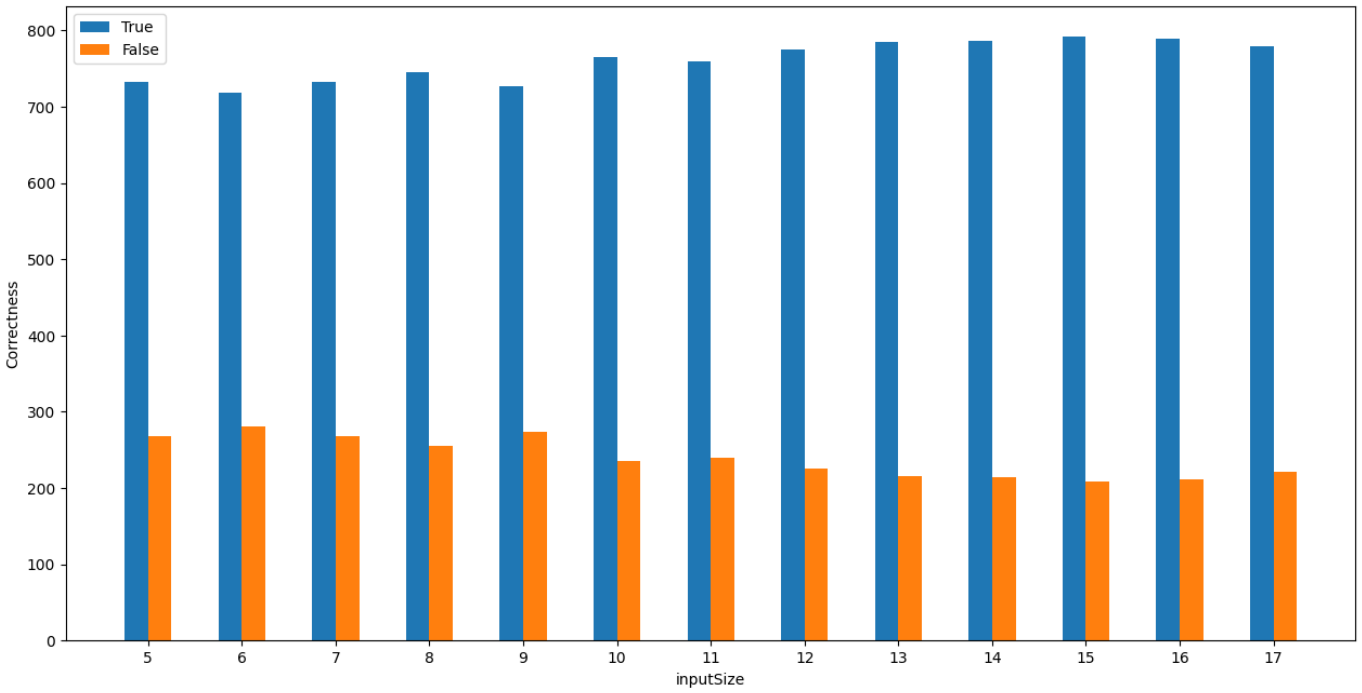


Slope of this line is approximately 2. Therefore, it can be said that the average running time is $\boldsymbol{\Theta(V^2)}$ because our running time changes as the input size gets bigger and the input is the number vertices of the graph**.**

The code for this analysis is as follows:

```
def test(size, k):
    start = time.time()
    graph = generator(size)
    vertex_cover_heuristic(graph, size, k)
    end = time.time()
    return end-start




def exec(N, size, k):
    tests = []
    for i in range(N):
        tests.append(test(size, k))
    # std = np.std(tests)
    mean = np.mean(tests)
    # sm = std/math.sqrt(N)
    # t = 1.645
    return mean

# graph for input sizes = [5,6,7,8,9,10]

inputSize = [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19, 20]
results = []
for i in range(5, 21):
    results.append(exec(100000, i, random.randint(1, i)))


xlog = np.log(inputSize)
ylog = np.log(results)
slope, intercept = np.polyfit(xlog, ylog, 1)
plt.loglog(inputSize, results)
print(slope)
plt.xlabel("log(inputSize)")
plt.ylabel("log(Running time mean)")
plt.show()
```

7. Experimental Analysis of the Quality



As it can be seen from the given chart, tests were done between the input sizes of 5 to 17 where each of them tested for 1000 times. It can be concluded that the correctness of the heuristic algorithm tends to increase as the input size grows.

The code for this analysis is as follows:

```
def qualityTest(inputSize):
    res = {
        "true": 0,
        "false": 0
    }
    for i in range(1000):
        graph = generator(inputSize)
        k = random.randint(1, inputSize)
        brute = vertex_cover_brute(graph, inputSize, k)
        heuristic = vertex_cover_heuristic(graph, inputSize, k)
        if heuristic[0] == brute[0]:
```

```
            res["true"] += 1
        else:
            res["false"] += 1
    return res


# test one inputSize for 100000 times

inputSize = [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
results = []

for i in range(5, 18):
    results.append(qualityTest(i))
    print(results)

true = []
false = []
for l in results:
    true.append(l["true"])
    false.append(l["false"])


barWidth = 0.25
fig = plt.subplots(figsize=(12, 8))
br1 = np.arange(len(true))
br2 = [x + barWidth for x in br1]
plt.bar(br1, true, width=barWidth, label='True')
plt.bar(br2, false, width=barWidth, label='False')
plt.xlabel('inputSize')
plt.ylabel('Correctness')
plt.xticks([r + barWidth/2 for r in range(len(true))], inputSize)
plt.legend()
plt.show()
```

8. Experimental Analysis of the Correctness (Functional Testing)

Black Box Testing (for edge cases):

- **Empty graph:**
  - Input: vertex_cover_heuristic([], 0, 1)
  -
  - Return: [False, []]
- **Empty edge set:**
  - Input: vertex_cover_heuristic([], 5, 1)
  - Return: [False, []]
- **Negative k value:**
  - Input: vertex_cover_heuristic([(1,2), (3,5), (2,4)], 5, -1)
  - Return: [False, []]

White Box Testing:

- **Statement coverage:**
  - **Test Case 1:** vertex_cover_heuristic([(1, 4), (1, 5), (2, 4), (2, 5)], 5, 3)
    - Return: [False, [1, 4, 2, 5]]
    - This test case covers the red highlighted parts
  - **Test Case 2:** vertex_cover_heuristic([(2, 3), (3, 4), (4, 5)], 5, 3)
    - Return: [True, [3, 4]]
    - This test case covers the green highlighted parts extra to the first test case
  - **Test Case 3:** vertex_cover_heuristic([(2, 3), (3, 4), (4, 5)], 5, 3)
    - Return: [False, []]
    - This test case covers purple highlighted parts

With these 3 test cases 100% statement coverage is achieved.

```
def remove(edges, u, v):
    newEdges = []
    for edge in edges:
        if (u not in edge) and (v not in edge):
            newEdges.append(edge)
    return newEdges
```

```
def vertex_cover_heuristic(edges, n_vertices, k):
    if k < 0 or k > n_vertices or n_vertices == 0 or edges == []:
        return [False, []]
    vertexCover = []
    UC = edges
    while len(UC) > 0:
        CE = UC[random.randint(0, len(UC)-1)]
        vertexCover.append(CE[0])
        vertexCover.append(CE[1])
        UC = remove(UC, CE[0], CE[1])

    if len(vertexCover) <= k:
        return [True, vertexCover]
    else:
        return [False, vertexCover]
```

9. Discussion

After doing all the analysis above for our code it can be concluded that our heuristic algorithm is working at 75% correctness when compared to the actual results. Moreover, it was concluded that this algorithm is **$\Theta(E)$** from theoretical analysis and **$\Theta(V^2)$** from the performance analysis. These two results support each other because it is known that **$\Theta(V^2) = \Theta(E)$** from **$V^2 = E$** in a dense graph. To conclude, our theoretical and experimental results do not contradict each other.

Submission

On SUCourse, for each group, only ONE person will submit the report.

For Progress Reports, we expect a report as PDF file. Your filename must be in the following format:

`CS301_Project_Progress_Report_Group_XXX.pdf`

where `XXX` is your group number.

For Final Reports, we expect a report as a PDF file, and also the codes and the data produced during the experiments. Please submit a single zip file that contains all your project files. Your zip file name must be in the following format:

`CS301_Project_Final_Report_Group_XXX.zip`

where `XXX` is your group number. In this zip file, please make sure that your final report is named as follows:

```
CS301_Project_Final_Report_Group_XXX.pdf
```

where `XXX` is your group number.