Approximate Counting

Gil Lopes Teixeira

*Resumo* – Este relatório apresenta dois métodos não determinísticos de contagem de caracteres em texto e compara-os com um Contador deterministico. O primeiro método apresentado apenas efetua uma contagem com probabilidade, fixa, de 25%. O segundo é um Contador com probabilidade descrescente com um fator:

Sendo k o total de contagens já efetuadas.

*Abstract* – This report presents two non deterministic methods to count characters in text and compares them to a deterministic counter. The first method is a fixed probability counter that counts every character with the same probability of 25%. The second method is a decreasing probability counter with a lesser factor of:

k being the total amount of times it already counted.

# I. Introduction

Non deterministic counters are very useful when working with big data. They produce smaller data structures that still contain most of the information and that are very useful for statistic analysis.

# II. Fixed Probability Counter

This counter counts with a, fixed, probablity of 25%. After a formal analisys of the algorithm we will compare the number of comparations, effective counts and expected counts.

def count\_fixed\_prob(self, fp, prob=0.25):

global comps

index = {}

for line in fp.readlines():

for word in line.split(' '):

for char in word:

to\_count = random.random()

if to\_count <= prob:

if char in index.keys():

index[char] += 1

else:

index[char] = 1

comps += 1

comps += 1

return index

For a given file with n characters excluding spaces the algorithm does:

k being the number of times the counter counted. Since the probability is fixed at ¼:

So the expected number of comparisons is:

|  |  |  |
| --- | --- | --- |
| Char | Expected Count | Exact Count |
| E | 23188.68 | 23186 |
| S | 11880.756 | 11874 |
| I | 10429.888 | 10429 |
| N | 10388.892 | 10390 |
| U | 10048.436 | 10053 |
| A | 9965.14 | 9958 |
| R | 9893.668 | 9904 |
| O | 9467.252 | 9465 |
| T | 9437.312 | 9438 |
| L | 8096.036 | 8093 |
| M | 4690.24 | 4693 |
| C | 4516.132 | 4516 |
| D | 4314.572 | 4315 |
| \n | 4180.996 | 4185 |
| P | 4116.256 | 4116 |
| V | 3108.732 | 3113 |
| , | 2737.78 | 2738 |
| É | 2437.112 | 2441 |
| - | 2388.788 | 2389 |
| . | 2263.656 | 2262 |
|  | 2057.632 | 2056 |
| Q | 1987.476 | 1987 |
| F | 1729.368 | 1734 |
| G | 1333.528 | 1333 |

Top 25 characters in the french version of the book from Shakespears book Winter Tales.

The expected count was calculated by multiplying the average count of each character, in 1000 tests, by 4.

# III. Greedy Algorithm

Calculating the maximum independent set of a Graph G with n vertexs:

current\_vertex = getMinEdgesVertex(graph, graph.vertexs)

res = []

vertexs\_to\_explore = (findNonAdjecentVertexes(graph, [current\_vertex]))

res.append(current\_vertex)

while True:

if not vertexs\_to\_explore:

break

current\_vertex = getMinEdgesVertex(graph, vertexs\_to\_explore)

res.append(current\_vertex)

vertexs\_to\_explore = findNonAdjecentVertexes(graph, res)

return res

This algorithm always chooses, in the next step, the valid\* vertex with minimum adjacent vertexs.

\*A valid vertex is a vertex that is not adjacente to the vertexs already choosen.

Let’s represente the length of the vertexs chosen list as v\_len

getMinEdgesVertex():

FindNonAdjacentVertexs(graph g, vertexs\_list v\_list):

Returns a subset of g containing vertexs that are not adjacente to any of the vertexs in v\_list.

Com v1 = len(v\_list)

findAdjacentVertexs(graph g, vertex v):

Worst Case:

Graph G with n vertexs and edges:

**Conclusion**

Calculating the maximum independent set:

Graph G with n vertexs and edges:

findAdjacentVertexs with 0 edges represents 0.

findNonAdjacentVertexs(x):

,

x being the size of the set of

vertexs already selected!

So:

For a graph G with edges:

Auxiliary functions:

getMinEdges(i) =

findAdjacentVertexs(i) = n\*(n-1)´

findNonAdjacentVertexs(i) = v\_len \*findAdjacentVertexs() + 2 \* n

Result:

=

Complexity:

Test Results:

|  |  |  |
| --- | --- | --- |
| n | Greedy | n^2 |
|  |  |  |
| 1 | 2 | 1 |
| 2 | 8 | 4 |
| 3 | 29 | 9 |
| 4 | 38 | 16 |
| 5 | 62 | 25 |
| 6 | 92 | 36 |
| 7 | 217 | 49 |
| 8 | 451 | 64 |
| 9 | 595 | 81 |
| 10 | 1052 | 100 |
| 11 | 940 | 121 |
| 12 | 1564 | 144 |
| 13 | 1862 | 169 |
| 14 | 3849 | 196 |
| 15 | 2650 | 225 |

# IV. Conclusion

The implementation of the exhaustive algorithm seems to be a fairly good one and the results of the number of comparisons obtained are very close to the estimated ones within the formal analysis presented. The greedy algorithm, though it did amount to about half the complexity of the exhaustive algorithm, doesn’t match with the expected results probably due to a sloppy implementation.

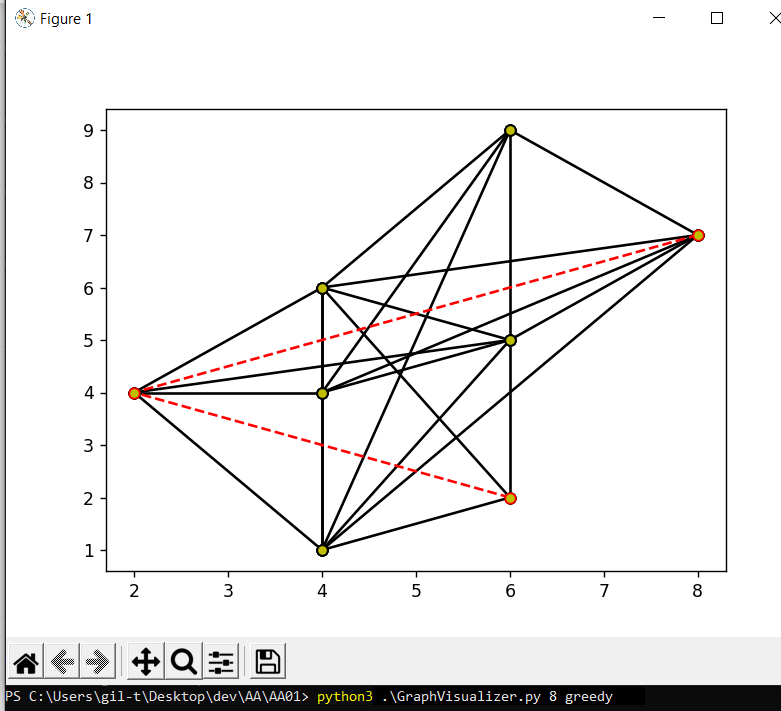


Figure 1 – Greedy Search for a 8 vertexs Graph

For this same exemple the exhaustive algorithm managed to find a maximum independent set of size 4 wich confirms that the greedy algorithm does not always find the optimal solution! Included in the readme.md file is a refence on how to generate this same graphs that were used. As long as the seed is the same so will be the graphs.