Instituto Politécnico Nacional

Escuela Superior de Cómputo

Desarrollo de un sistemas distribuidos

Optimizacion

Grupo: 4CM1

Integrantes:
Angel Lopez Manriquez
Aldo Suárez cruz
Mónica Nataly Toxtli Calderón
Irving Arturo Aguiar Hernández

Profesor:
Coronilla Contreras Ukranio

Fecha de realización: 14 de mayo de 2020

Reporte de practica

4CM1

14 de mayo de 2020

Índice

1. Parte

1.1. Tiempo usado para 10k votantes

El servidor no se preocupa si hay votos repetidos.

```
(base) aingeru@gugul:~/Documents/git/dist/15Optimizacion$ time ./client

real    0m6.790s
user    0m0.160s
sys     0m0.514s
```

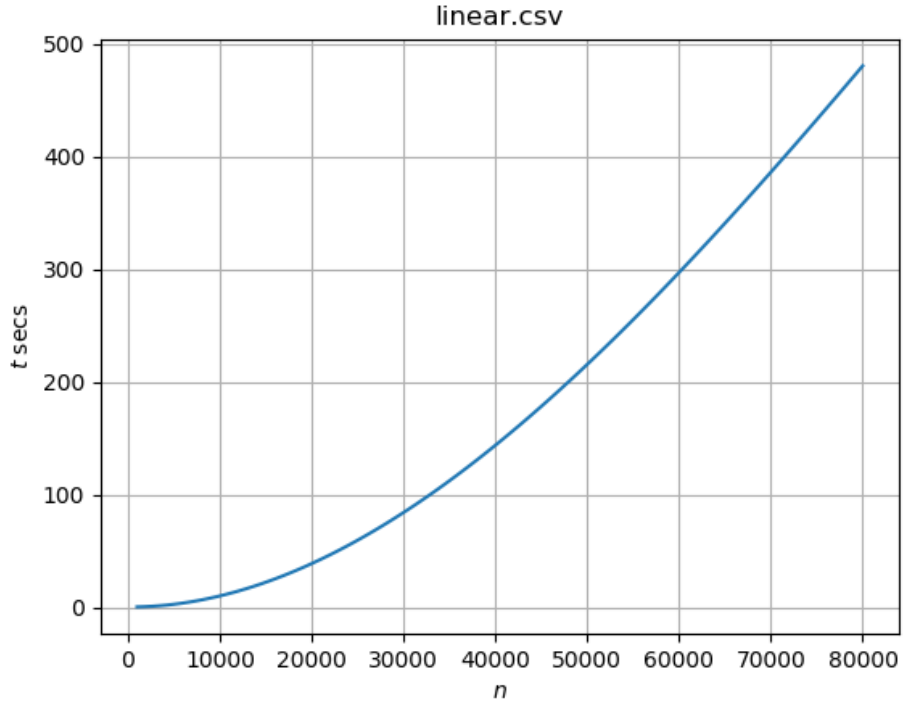
1.2. Tiempo usado para 10k votantes

El servidor se preocupa si hay votos repetidos, usa busqueda lineal.

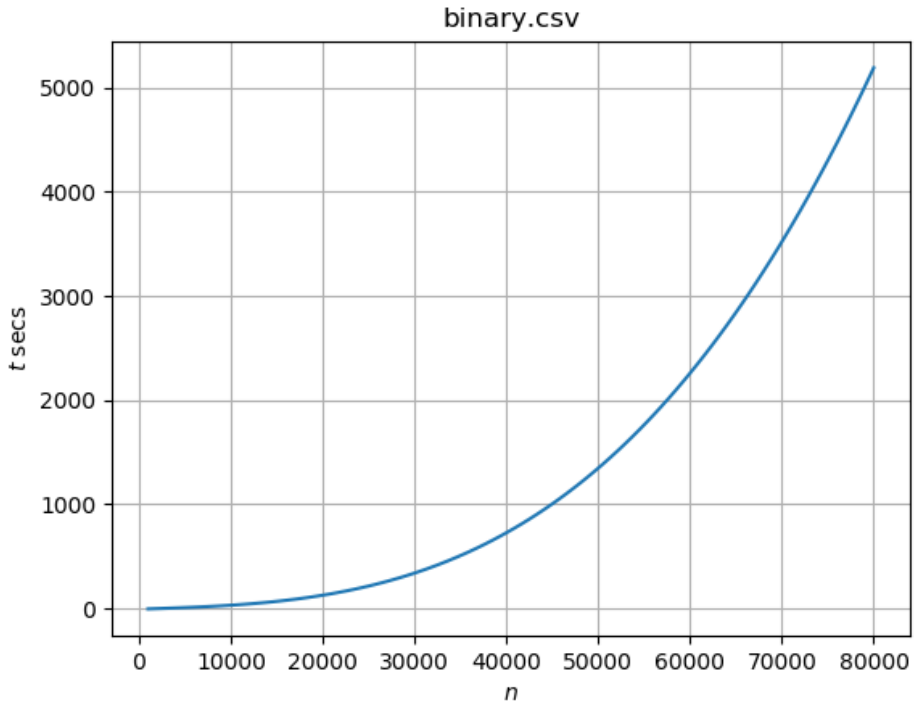
```
(base) aingeru@gugul:~/Documents/git/dist/15Optimizacion$ time ./client 10000

real    0m10.049s
user    0m0.164s
sys     0m0.589s
```

1.3. Preguntas

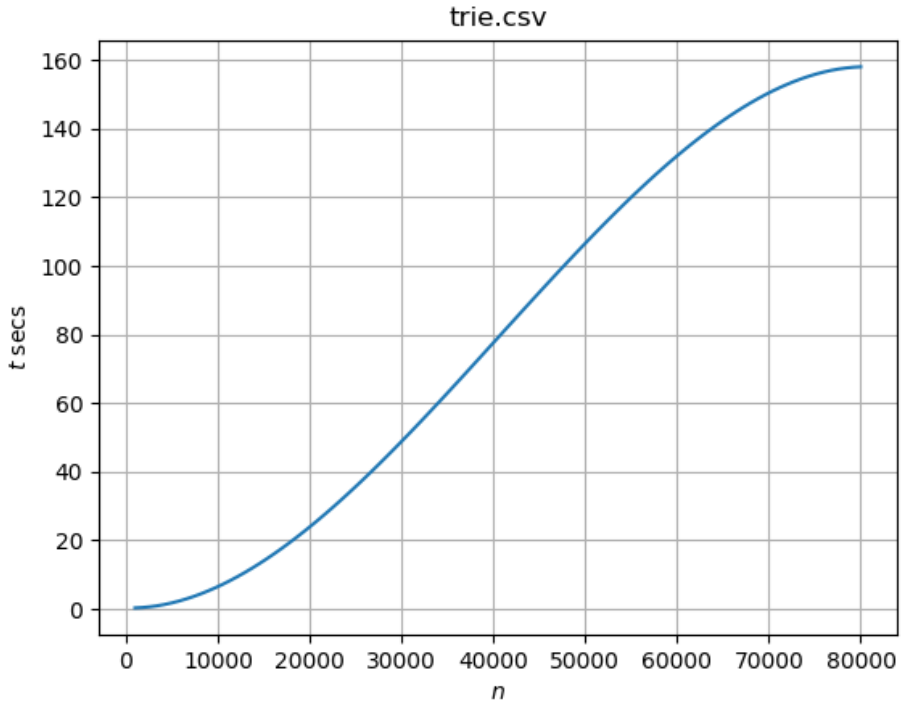
Las siguientes graficas son predicciones usando polinomios de Lagrange ajustado con $n = 5000, 10000, 15000, 20000$, aunque una desventaja de este metodo de aproximiacion es que el error de la prediccion es mayor a medida que x de prueba se aleja de x de entrenamiento.



La busqueda lineal toma un tiempo considerablemente adecuado, aunque claro que podria ser mejorado.



Se aprecia que el peor tiempo lo toma la busqueda binaria, y es de esperarse ya que para poder realizar la busqueda el vector tiene que estar ordenado, lo cual agrega mas tiempo computacional. Como se sugiere podemos usar un Trie o en mi opinion una prority queue para evadir el ordenamiento cada vez que se busca por el registro.



La estructura de datos Trie es la mas optimizada para este problema pues es mejor que la busqueda lineal en tiempo, y gracias a que la clave puede ser indexada y no tenemos que ordenar cada vez que preguntamos por un elemento.

Como dice la teoria, las operaciones de insercion y busqueda sobre la estructura Trie son de orden lineal. A diferencia de la busqueda lineal se usa menos almacenamiento y las consultas son mucho mejor que consultar la posicion i de un vector. El analisis del algoritmo dicta que se llevarian a lo mas 70 000 0000 segundos, aproximadamente 810.18 dias o 2.21 años.

2. Parte 2

Se hizo la implementacion de la estructura usando arboles n-arios.

```
1 /** Trie implementation by using KTrees, it maps string -> bool.
2  *
3  */
4
5 #ifndef __TRIE_H__
6 #define __TRIE_H__
```

```
7
8  #include <iostream>
9  #include <algorithm>
10 #include <iterator>
11 #include <vector>
12 #include <memory>
13 #include <string>
14
15
16 using namespace std;
17
18 template<typename T>
19 struct Trie {
20
21     struct Node {
22         char character;
23         T data;
24         std::vector<std::unique_ptr<struct Node > > children;
25
26         Node() {}
27         Node(char &_char): character(_char) { }
28         Node(char &_char, T _data): character(_char), data(_data) { }
29     };
30
31
32     using u_ptr_node = std::unique_ptr<Node >;
33     using u_ptr_nodes = std::vector<u_ptr_node >;
34
35     std::unique_ptr<Node > root;
36
37
38     Trie() {
39         root = std::make_unique<Node >();
40     }
41
42     void __put(std::string &key, unsigned i, T &value, u_ptr_node &current_ptr)
43     {
44         auto &children = current_ptr.get()->children;
45         // Insert the leaf with the value associated
46         // If the element was inserted previously it'll be inserted again
47         if (i == key.size() - 1) {
48             children.push_back(make_unique<Node> (key[i], value));
49             return;
50         }
51         auto it = std::find_if(children.begin(), children.end(),
52                 [&](u_ptr_node &child) -> bool {
53                     return (child.get()->character == key[i]);
54         });
55         if (it != children.end()) { // found
```

```
56              __put(key, i + 1, value, *it);
57              return;
58          }
59          children.push_back(make_unique<Node> (key[i]));
60          __put(key, i + 1, value, children[children.size() - 1]);
61      }
62
63      void put(std::string &key, T value) {
64          __put(key, 0, value, root);
65      }
66
67      Node * __get_node(u_ptr_node &self, std::string &key, unsigned i) {
68          if (i >= key.size())
69              return nullptr;  // index out of bounds
70          auto &children = self.get()->children;
71          auto it = std::find_if(children.begin(), children.end(),
72                  [&] (auto &node_uptr) -> bool {
73                      return node_uptr.get()->character == key[i];
74          });
75          if (it == children.end())
76              return nullptr; // No such value
77          if (i == key.size() - 1)
78              return it->get();
79          return __get_node(*it, key, i + 1);
80      }
81
82      T get(std::string &key) {
83          Node *node_ptr = __get_node(root, key, 0);
84          if (node_ptr == nullptr)
85              throw "Trie:get: There's no such value\n";
86          return node_ptr->data;
87      }
88
89      T get(std::string &key, T &default_value) {
90          Node *node_ptr = __get_node(root, key, 0);
91          if (node_ptr == nullptr)
92              return default_value;
93          return node_ptr->data;
94      }
95
96      bool has(std::string key) {
97          return __get_node(root, key, 0) != nullptr;
98      }
99
100 };
101
102
103 #endif
```

Las pruebas descritas anteriormente fueron hechas usando esta estructura.

Cada entrada a la estructura Trie ocupa al menos 11 bytes como clave (`char [11] celular`) y 16 como valor (`2 sizeof(long)`). Asi tenemos que la funcion de espacio seria aproximadamente $\Theta(n) = 16n$ por lo que con 10 000 votos tenemos 160 KB y con 70 M votos tenemos 1.12 Gb en ram.