

FACULTAD DE INGENIERÍA
UNIVERSIDAD DE BUENOS AIRES

TEORÍA DE ALGORITMOS

Trabajo Práctico N1

Autores:

Addin Kevin

Cabrera Jorge

Gatti Nicolas

Orlando Juan Manuel

Padrón:

94280

93310

93570

93152

14 de Octubre, 2016



Contents

1	Estadístico de orden k	3
1.1	Fuerza bruta	3
1.1.1	Implementación	3
1.1.2	Complejidad	3
1.1.3	Mejor y peor caso	4
1.1.4	Gráficos	4
1.2	Ordenar y seleccionar	6
1.2.1	Implementación	6
1.2.2	Complejidad	6
1.2.3	Mejor y peor caso	6
1.2.4	Gráficos	7
1.3	K-Selecciones	9
1.3.1	Implementación	9
1.3.2	Complejidad	10
1.3.3	Mejor y peor caso	10
1.3.4	Gráficos	11
1.4	K-Heapsort	13
1.4.1	Implementación	13
1.4.2	Complejidad	13
1.4.3	Mejor y peor caso	13
1.4.4	Gráficos	14
1.5	HeapSelect	16
1.5.1	Implementación	16
1.5.2	Complejidad	16
1.5.3	Mejor y peor caso	16
1.5.4	Gráficos	17
1.6	QuickSelect	19
1.6.1	Implementación	19
1.6.2	Complejidad	20
1.6.3	Mejor y peor caso	20
1.6.4	Gráficos	21
1.7	Conclusiones	23
2	Recorridos en grafos	25
2.1	BFS	25
2.1.1	Implementación	25

2.1.2	Complejidad	30
2.1.3	Mejor y peor caso	30
2.1.4	Gráficos	31
2.2	Dijkstra	33
2.2.1	Implementación	33
2.2.2	Complejidad	34
2.2.3	Mejor y peor caso	34
2.2.4	Análisis	35
2.3	Búsqueda con heurística	37
2.3.1	Implementación	37
2.3.2	Complejidad	38
2.3.3	Mejor y peor caso	38
2.3.4	Análisis	40
2.4	A*	43
2.4.1	Implementación	43
2.4.2	Complejidad	44
2.4.3	Mejor y peor caso	44
2.4.4	Análisis	44

1 Estadístico de orden k

1.1 Fuerza bruta

1.1.1 Implementación

```
1 bool Utils::verificador(vector<int> array, int candidate, int k)↵
2 {
3     int leftCount = 0;
4     int frequency = 0;
5     for (int i = 0; i < array->size(); i++) {
6         if (array->at(i) < candidate) {
7             leftCount++;
8         } else if (array->at(i) == candidate) {
9             frequency++;
10        }
11    }
12    return leftCount <= k && k <= leftCount + frequency - 1;
13}
14
15
16 int Utils::bruteForce(vector<int> array, int k){
17     for(int i=0; i < array->size(); i++){
18         if(verificador(array, array->at(i), k))
19             return array->at(i);
20     }
21     return -1;
22 }
```

1.1.2 Complejidad

Para analizar la complejidad de este algoritmo, en primer lugar debemos observar que tenemos un ciclo for que puede llegar a recorrer los $n = array->size()$ valores del arreglo. En cada una de esas iteraciones, se ejecuta el método verificador, para verificar si ese valor es el elemento k del arreglo.

El método verificador lo que hace es recorrer el arreglo, por lo que tiene una complejidad $\Theta(n)$.

Por lo tanto, tenemos n (tamaño del arreglo) iteraciones, donde en cada iteración se realizan $O(n)$ operaciones.

$$T(n) = O(n^2)$$

1.1.3 Mejor y peor caso

Para este algoritmo, lo mejor que nos puede pasar es que hagamos una sola iteración. Esto ocurriría si tenemos un array, por ejemplo, $A = 1, 5, 9, 4, 3, 8, 2, 10$. Si lo que queremos es $k = 0$, haremos una única iteración preguntando si el valor 1 es el "k=0" del array. Este caso sería $\Theta(n)$.

Pero, si tenemos mala suerte, tendremos que hacer n iteraciones, cada una con costo de $\Theta(n)$. Usando el arreglo recién mencionado, esto pasaría si $k = 7$, dándonos una complejidad de $\Theta(n^2)$.

1.1.4 Gráficos

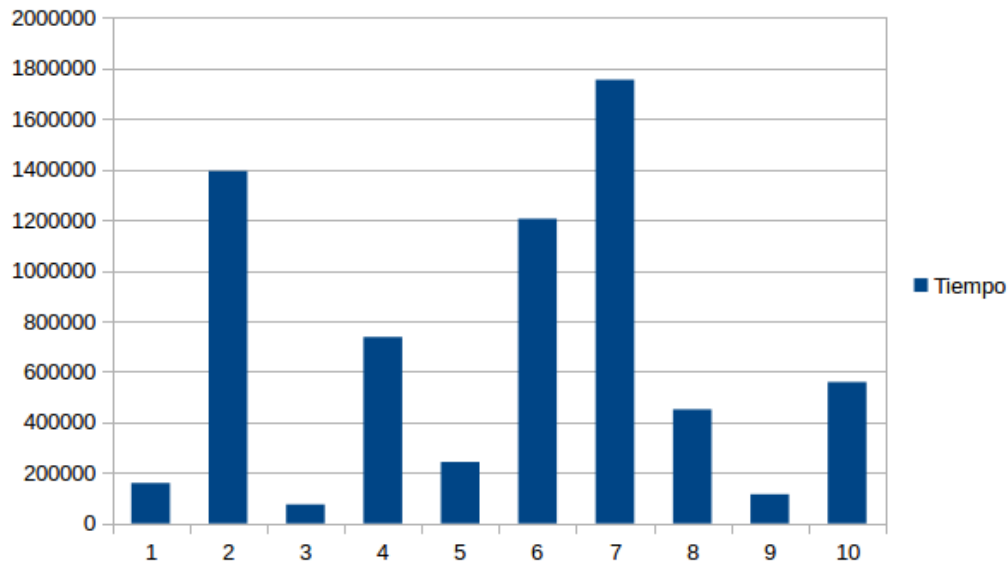


Figure 1: Fuerza bruta para $k = 0$, $n = 10000$

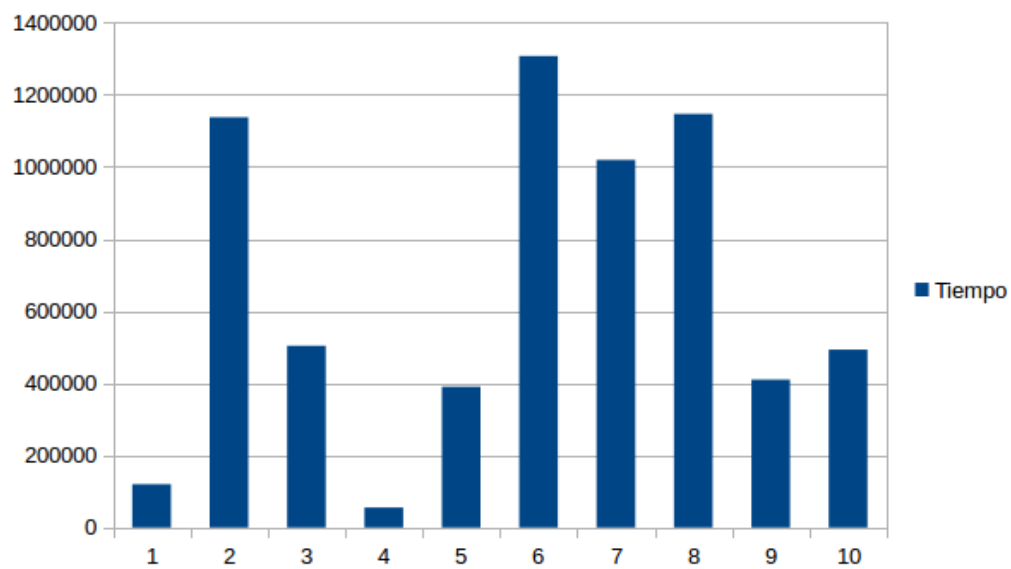


Figure 2: Fuerza bruta para $k = n/2$, $n = 10000$

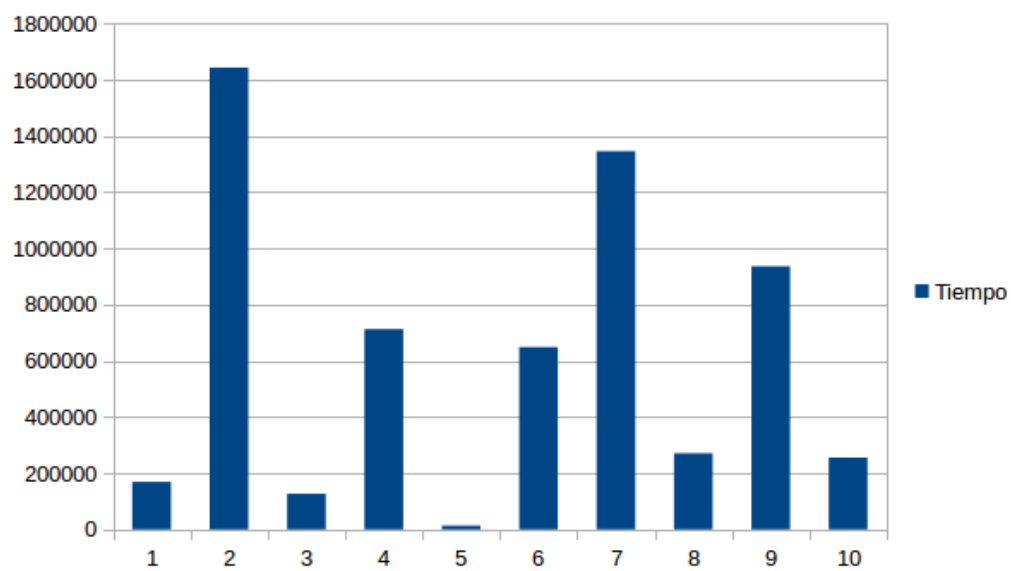


Figure 3: Fuerza bruta para $k = n-1$, $n = 10000$

1.2 Ordenar y seleccionar

1.2.1 Implementación

```
1 int Utils::orderAndSelect(vector<int> array, int k) {  
2     std::sort(array->begin(), array->end());  
3     return array->at(k);  
4 }
```

1.2.2 Complejidad

La complejidad del algoritmo radica en el algoritmo de ordenamiento. El algoritmo de ordenamiento tiene una complejidad $O(n\log(n))$ [1].

1.2.3 Mejor y peor caso

Para este algoritmo, el mejor y peor caso va a depender de cómo sea el algoritmo de ordenamiento. En todos los casos, para cualquier arreglo, se tiene que ordenar que tiene una complejidad de $O(n\log(n))$.

$$T(n) = O(n\log(n))$$

1.2.4 Gráficos

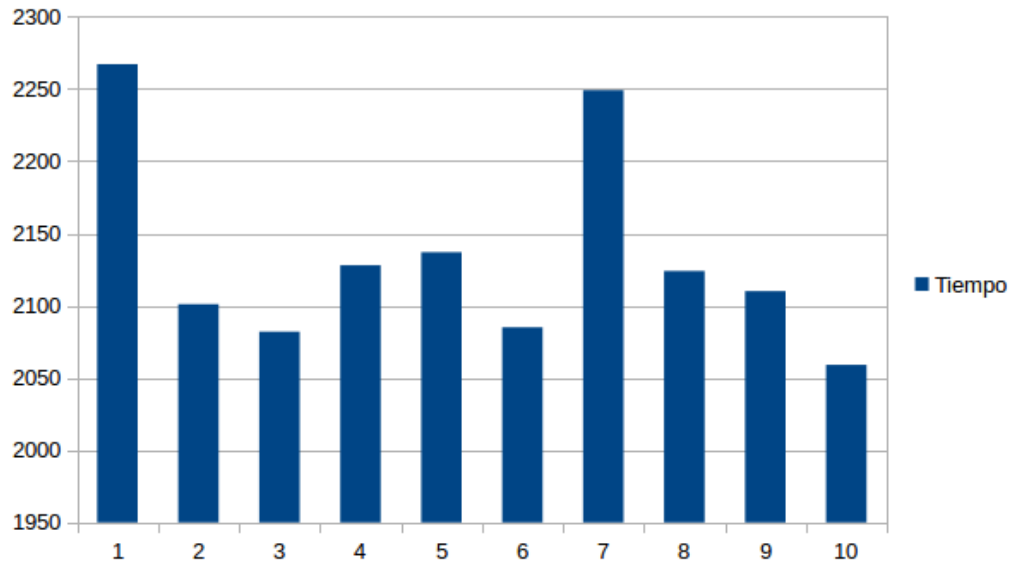


Figure 4: Ordenar y seleccionar para $k = 0$, $n = 10000$

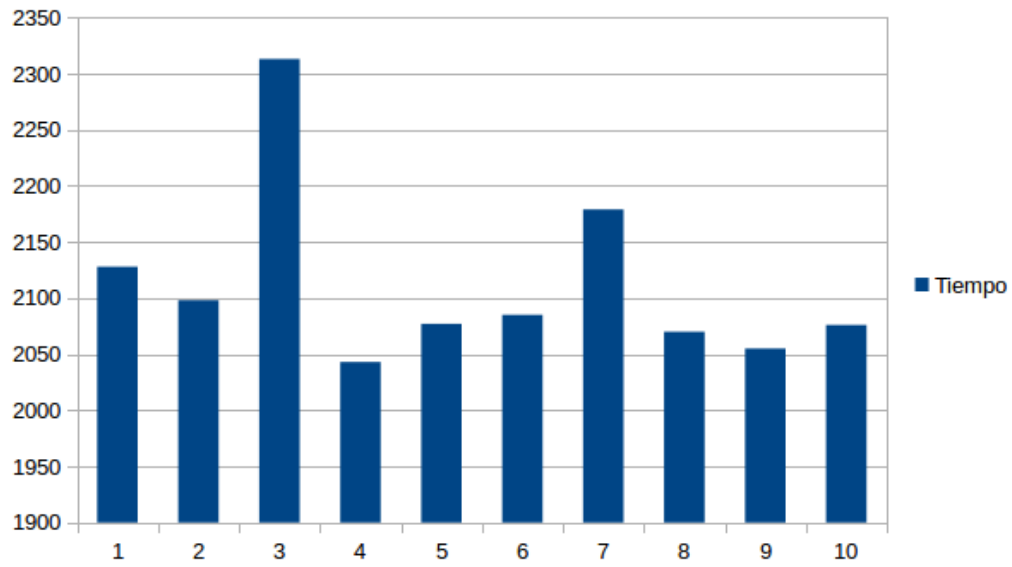


Figure 5: Ordenar y seleccionar para $k = n/2$, $n = 10000$

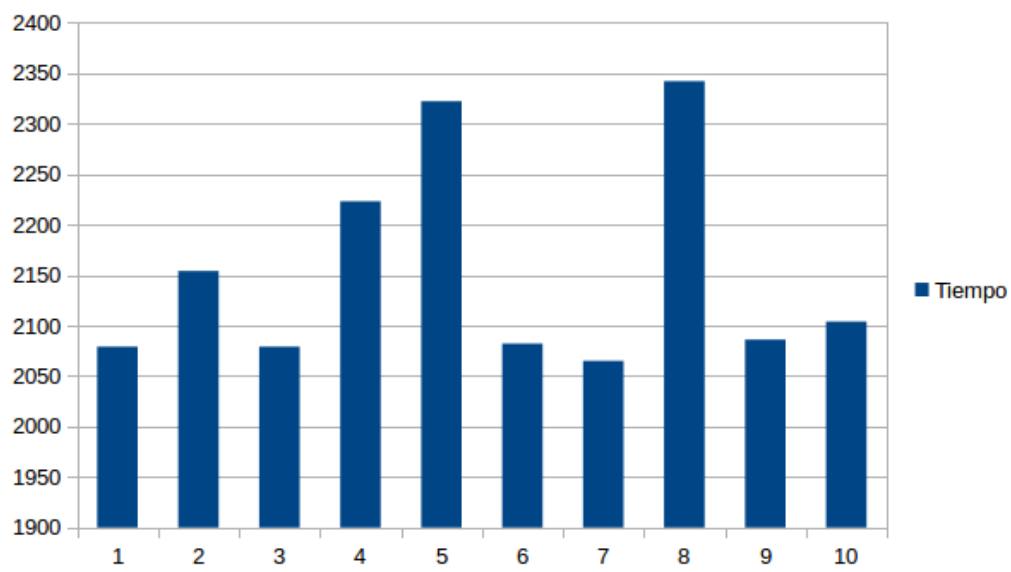


Figure 6: Ordenar y seleccionar para $k = n-1$, $n = 10000$

1.3 K-Selecciones

1.3.1 Implementación

```
1 / Busca la posicion del elemento mas chico , buscando a partir ↵
   de initPosition /
2 int Utils::getPositionOfSmallerValue(vector<int> array, int ↵
   initPosition){
3     int smallerPosition = initPosition;
4     int smallerValue = array->at(initPosition);
5     for(int auxPosition = (initPosition+1); auxPosition < array↵
       ->size(); auxPosition++){
6         if(array->at(auxPosition) < smallerValue){
7             smallerPosition = auxPosition;
8             smallerValue = array->at(auxPosition);
9         }
10    }
11    return smallerPosition;
12 }
13
14 / Swap entre initPosition y smallerPosition /
15 void Utils::changeSmallerWithInitial(vector<int> array, int ↵
   initPosition, int smallerPosition){
16     int initValue = array->at(initPosition);
17     int smallerValue = array->at(smallerPosition);
18
19     array->at(initPosition) = smallerValue;
20     array->at(smallerPosition) = initValue;
21 }
22
23 int Utils::kSelection(vector<int> array, int k){
24     for(int indexArray=0; (indexArray<array->size() && ↵
       indexArray <= k); indexArray++){
25         int position = getPositionOfSmallerValue(array, ↵
       indexArray);
26         changeSmallerWithInitial(array, indexArray, position);
27     }
28     return array->at(k);
29 }
```

1.3.2 Complejidad

En el algoritmo de K-Selecciones, tenemos un ciclo de k iteraciones. Dentro de dicho ciclo, lo que se hace es obtener la posición del elemento más chico, a partir de la posición indicada por *indexArray*. Este método realiza $O(n - i)$ operaciones, siendo n el tamaño del array y siendo i la posición a partir de la cual se busca.

El método "changeSmallerWithInitial" es un swap, que tiene un costo de $O(1)$.

Si hacemos las cuentas, nos quedaría que:

$$T(n) = n + (n - 1) + (n - 2) + \dots + (n - k) = \sum_{i=0}^k (n - i) = n(k + 1) - \frac{k(k + 1)}{2}$$

1.3.3 Mejor y peor caso

Analizando la complejidad calculada en el punto anterior, vemos que el mejor caso será cuando $k = 0$, quedándonos $\Theta(n)$. Esto es, si $A = [5, 3, 2, 1, 7, 8, 4, 9]$, si buscamos el elemento $k = 0$, recorreríamos todo el arreglo para encontrarlo.

El peor caso es cuando tenemos que hacer $k = n - 1$ selecciones. En este caso, lo que estaríamos haciendo sería ordenar el arreglo, de una forma incorrecta. Nos quedaría $\Theta(n^2)$.

1.3.4 Gráficos

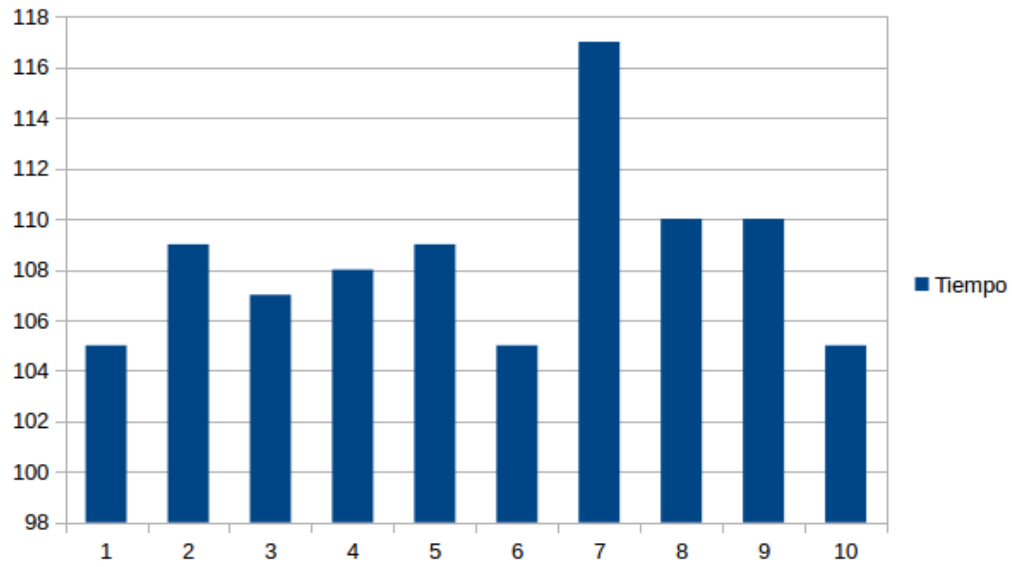


Figure 7: K-Selecciones para $k = 0$, $n = 10000$

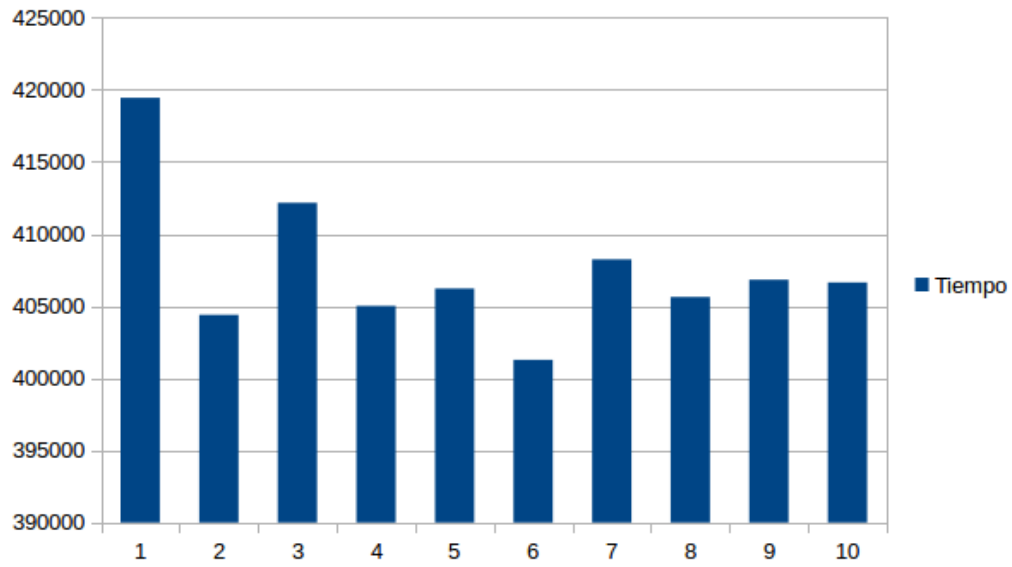


Figure 8: K-Selecciones para $k = n/2$, $n = 10000$

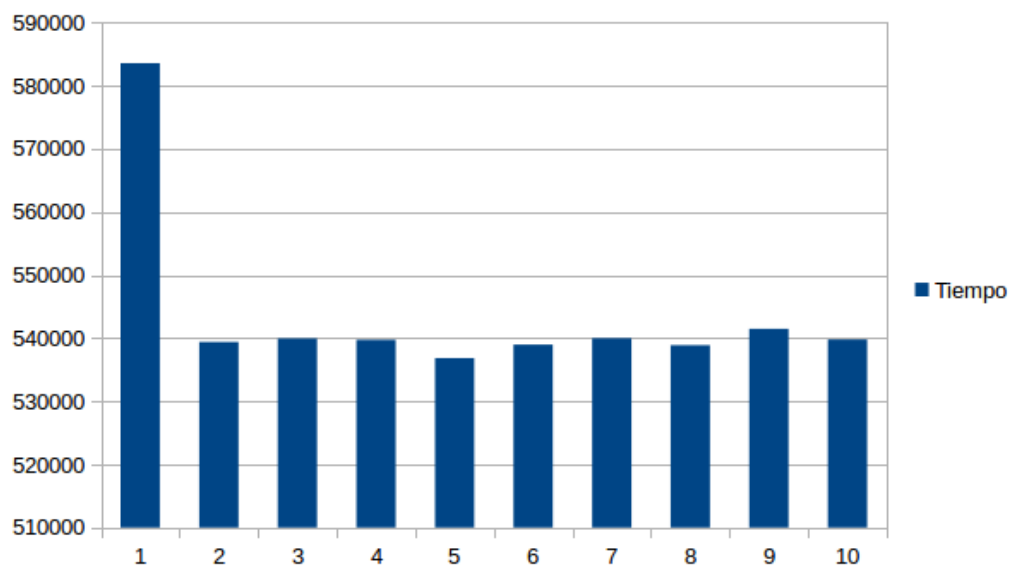


Figure 9: K-Selecciones para $k = n-1$, $n = 10000$

1.4 K-Heapsort

1.4.1 Implementación

```
1 int Utils::kHeapSort(vector<int> & array, int k) {  
2     std::make_heap(array->begin(), array->end(), std::greater<int>()); // Heapify  
3  
4     int extraccion;  
5     // K extracciones  
6     for (int i = 0; i < k + 1 ; i++) {  
7         std::pop_heap(array->begin(), array->end(), std::greater<int>());  
8         extraccion = array->back();  
9         array->pop_back();  
10    }  
11  
12    return extraccion;  
13 }
```

1.4.2 Complejidad

La primera operación que realizamos es un heapify. Este heapify tiene una complejidad $O(n)$ [2].

Luego tenemos un ciclo de k iteraciones, donde en cada ciclo se realiza una extracción del elemento más chico. Esto tiene una complejidad $O(\log(n))$ [3], ya que tenemos que mantener la propiedad de heap.

Por lo tanto nos queda que:

$$T(n) = O(n + k\log(n))$$

1.4.3 Mejor y peor caso

El mejor caso se daría cuando tenemos $k = 0$, por lo que tendríamos que hacer el heapify y luego realizar una única extracción, para cualquier tipo de arreglo. Esto es $O(n + \log(n)) = O(n)$.

Pero si tenemos que realizar $k = n - 1$ extracciones, para cualquier tipo de arreglo tendríamos $O(n + n\log(n)) = O(n\log(n))$.

1.4.4 Gráficos

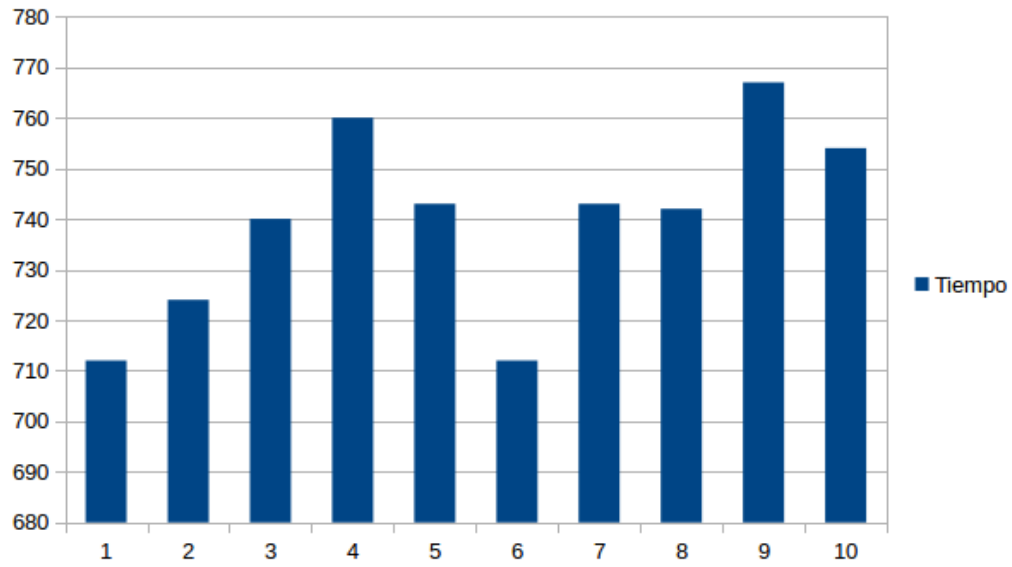


Figure 10: K-HeapSort para $k = 0$, $n = 10000$

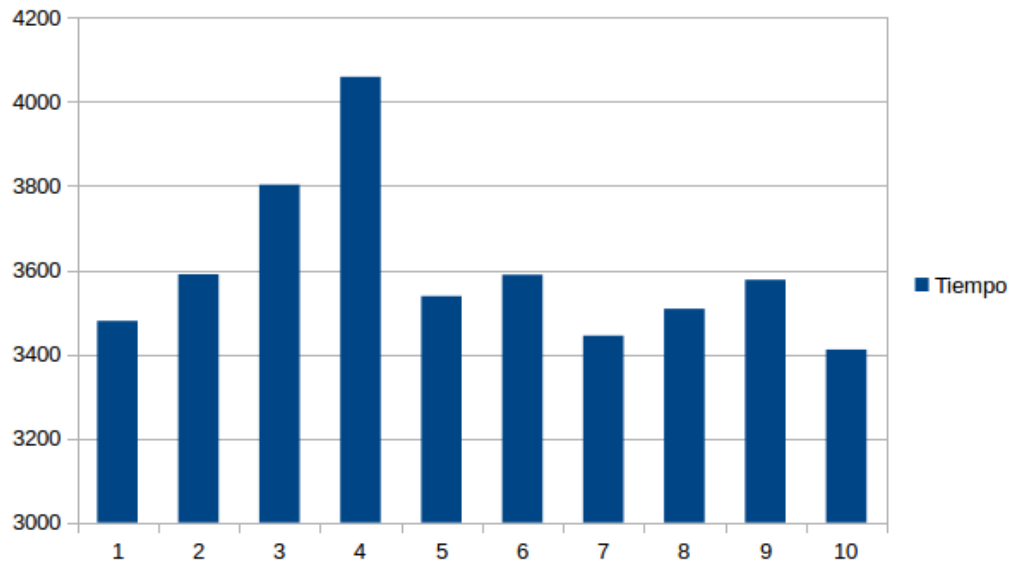


Figure 11: K-HeapSort para $k = n/2$, $n = 10000$

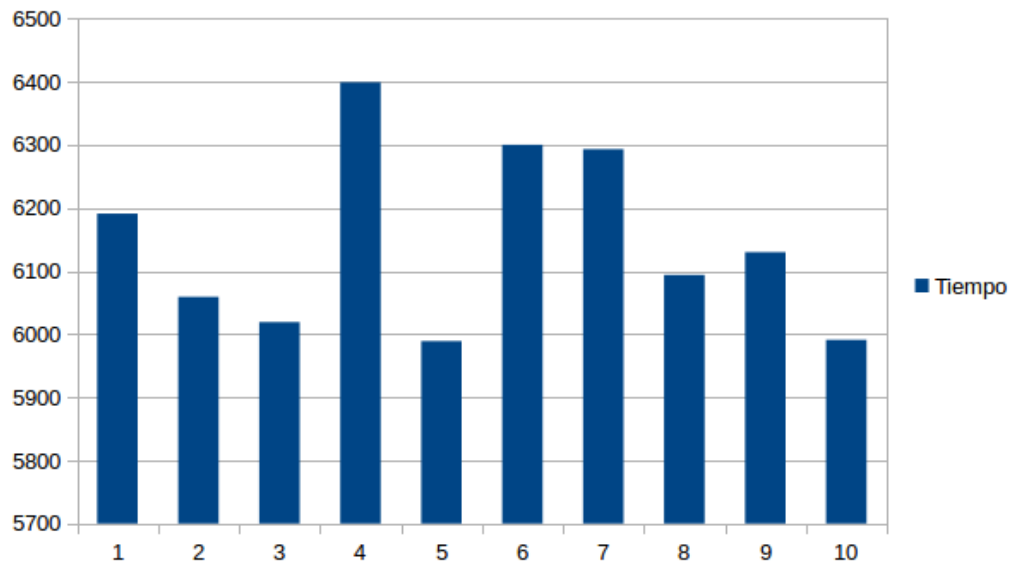


Figure 12: K-HeapSort para $k = n-1$, $n = 10000$

1.5 HeapSelect

1.5.1 Implementación

```
1 int Utils::heapSelect(vector<int> array, int k) {
2     std::vector<int> heapK(array->begin(), array->begin()+k+1);
3     std::make_heap(heapK.begin(), heapK.end()); // Heapify de ←
        los k primeros elementos.
4
5     for (int i = k+1 ; i < array->size(); i++) {
6         if (heapK.at(0) > array->at(i)) {
7             std::pop_heap(heapK.begin(), heapK.end());
8             heapK.pop_back();
9             heapK.push_back(array->at(i));
10            std::push_heap(heapK.begin(), heapK.end());
11        }
12    }
13
14    return heapK.at(0);
15 }
```

1.5.2 Complejidad

Lo primero que hacemos en el algoritmo es hacer una copia del arreglo de los k primeros elementos, y luego hacer heapify. Esto tiene un costo de $O(k)$ [2].

Luego tenemos un ciclo de $n - k$ iteraciones. Dentro de cada iteración, si el elemento máximo dentro del heap de tamaño k es más grande que el valor que se visita del resto del arreglo, se realiza el intercambio. Este intercambio consiste en sacar el elemento más grande, $O(\log(n))$ [3], y luego colocar el elemento más chico, $O(\log(n))$ [4].

Si agrupamos todo, tenemos que, considerando el caso que siempre se entre al if:

$$T(n) = O(k + (n - k)\log(n))$$

1.5.3 Mejor y peor caso

Tenemos dos posibles mejores casos para este algoritmo. El primero sería que en el heap de k elementos que nos queda al principio, sea el que tenga los k elementos más chicos. Por ejemplo sea, $A = [0, 4, 3, 2, 9, 10, 15, 13]$. Si queremos $k = 0$, el heap nos quedaría $H = [0]$. Al iterar por el resto del

arreglo, [4, 3, 2, 9, 10, 15, 13], ninguno de estos elementos entrará al if, por lo que solo tendremos un costo de $O(1)$ para cada iteración. En total, nos quedaría $T(n) = O(n)$.

El segundo caso sería si $k = n - 1$, el heap nos quedaría $H = [15, 13, 10, 4, 9, 0, 3, 2]$. Luego no tendríamos que realizar nada mas. Nos quedaría que $T(n) = O(n)$.

Para analizar el peor caso, supongamos que tenemos el array $B = [5, 10, 8, 9, 13, 1, 4, 0]$. Si queremos $k = 0$, el heap inicial nos quedaría $H = [5]$. Como el elemento más chico (0), está en la ultima posición vamos a entrar al if en todas las iteraciones, y como cada iteración es $O(\log(n))$, la complejidad total nos quedará $T(n) = O(n\log(n))$

1.5.4 Gráficos

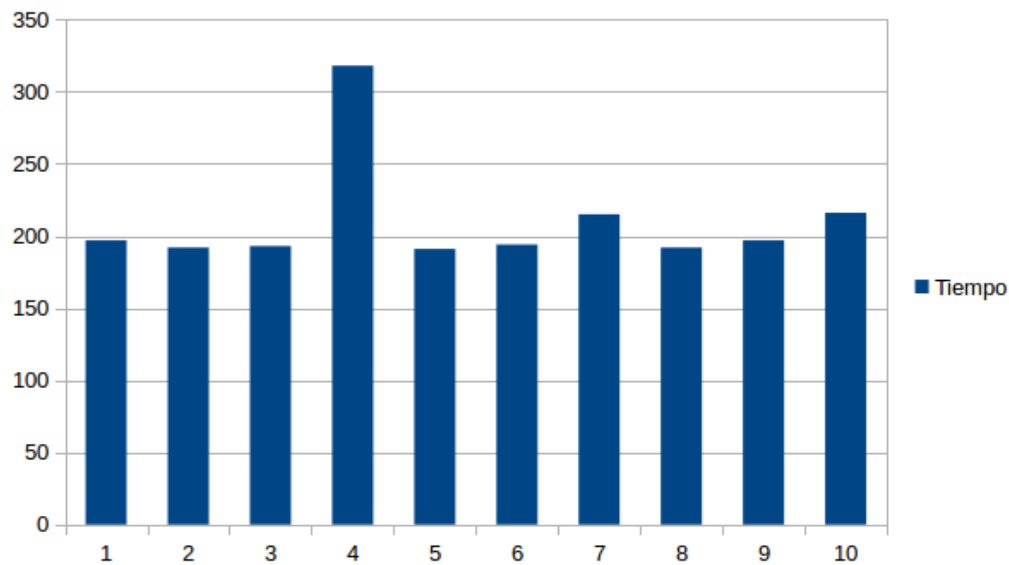


Figure 13: HeapSelect para $k = 0$, $n = 10000$

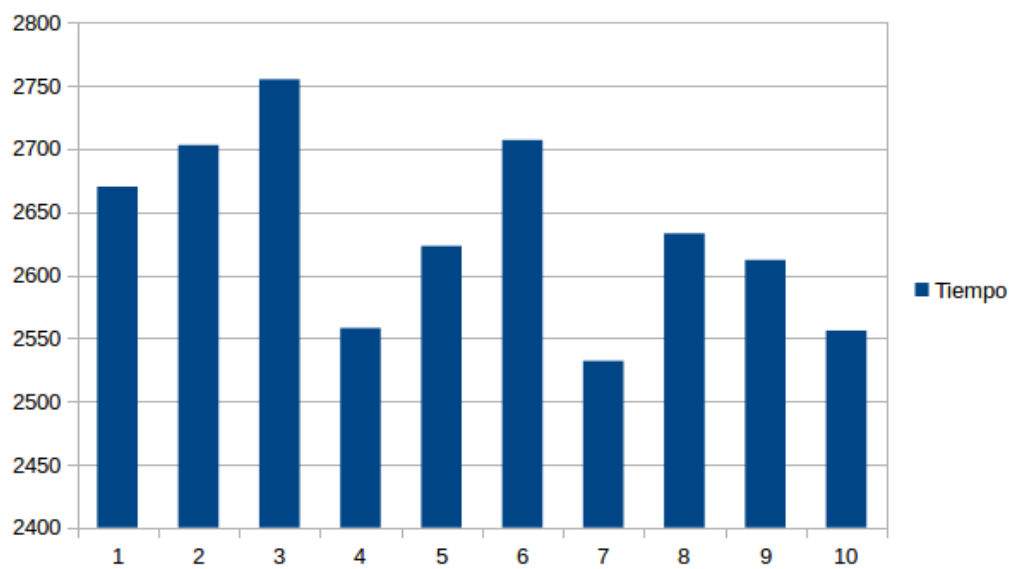


Figure 14: HeapSelect para $k = n/2$, $n = 10000$

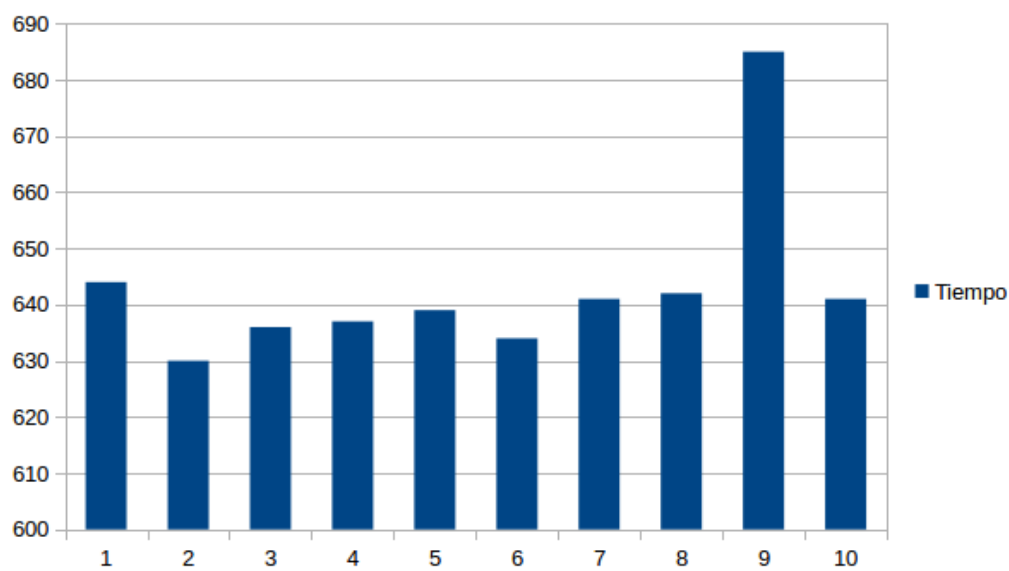


Figure 15: HeapSelect para $k = n-1$, $n = 10000$

1.6 QuickSelect

1.6.1 Implementación

```
1 void Utils::swap(vector<int> array, int i, int j) {
2     int tmp = array->at(i);
3     array->at(i) = array->at(j);
4     array->at(j) = tmp;
5 }
6
7 int Utils::partition(vector<int> array, int left, int right) {
8     int pivotIndex = left + std::rand() % (right - left);
9     int pivotValue = array->at(pivotIndex);
10
11     swap(array, pivotIndex, right); // Move pivot to end
12     int storeIndex = left;
13     for (int i = left; i < right; i++) {
14         if (array->at(i) < pivotValue) {
15             swap(array, storeIndex, i);
16             storeIndex++;
17         }
18     }
19     swap(array, right, storeIndex); // Move pivot to its final place
20     return storeIndex;
21 }
22
23 int Utils::_quickSelect(vector<int> array, int left, int right, ←
24     int k) {
25     if (left == right)
26         return array->at(left);
27
28     int pivotIndex = partition(array, left, right);
29
30     if (pivotIndex == k)
31         return array->at(pivotIndex);
32     else if (k < pivotIndex)
33         return _quickSelect(array, left, pivotIndex - 1, k);
34     else
35         return _quickSelect(array, pivotIndex + 1, right, k);
36 }
37
38 int Utils::quickSelect(vector<int> array, int k) {
39     return _quickSelect(array, 0, array->size() - 1, k);
40 }
```

1.6.2 Complejidad

La complejidad del algoritmo es [5]:

$$T(n) = \Theta(n)$$

1.6.3 Mejor y peor caso

El mejor y el peor caso va a depender del pivote. En nuestra implementación utilizamos un pivote al azar entre $[left, right]$, lo que disminuye la probabilidad de elegir un mal pivote.

Como en quicksort, el quickselect tiene una buena performance promedio, pero es sensible al pivote que se elija. Ejemplo, supongamos que $A = [0, 1, 2, 3, 4, 5, 6]$. Al realizar la partición, si elegimos siempre el pivote del extremo izquierdo, en este caso 0, la reducción del arreglo va a ser de a un elemento. Por lo que nos quedará $O(n^2)$. Esto es porque en la primera partición se realiza n comparaciones, en la segunda partición $n - 1$, en la tercera partición $n - 2$, llevándonos a que la complejidad del algoritmo sea $O(n^2)$, ya que la partición nos disminuye el espacio de búsqueda en solo una unidad. Cuando se eligen buenos pivotes, esto provoca que el espacio de búsqueda se vaya reduciendo, siendo el algoritmo $O(n)$ [9]

Si contamos con suerte, y si se elije el pivote correcto desde el principio, podemos hacer que se hagan solamente n comparaciones.

Ejemplo, $B = [5, 3, 6, 9, 10, 15]$.

Si queremos $k = 3$, y tomamos de pivote 9, luego de hacer la partición (en la que haremos n comparaciones) terminaría el algoritmo. Este caso nos daría el mejor tiempo posible.

1.6.4 Gráficos

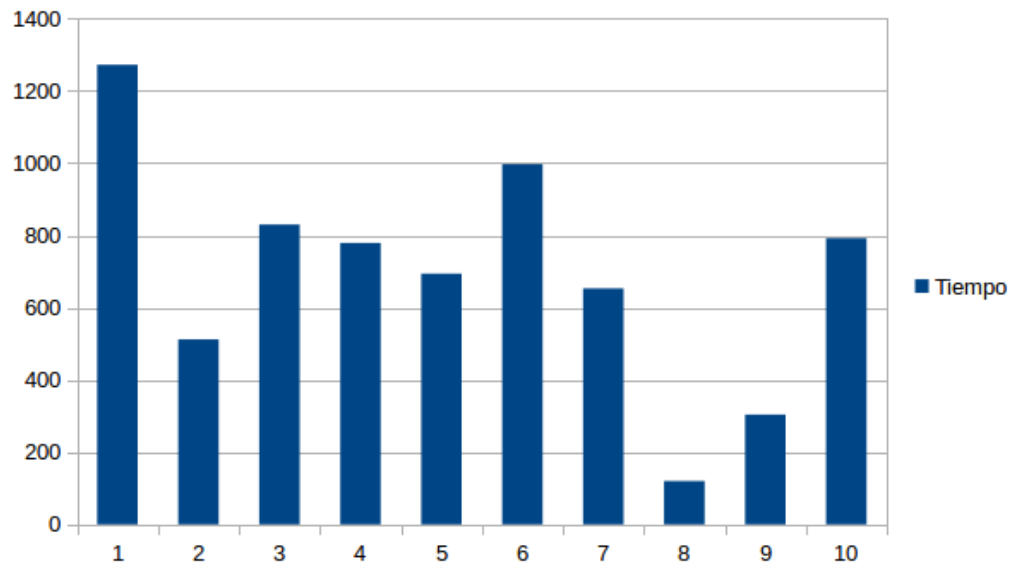


Figure 16: QuickSelect para $k = 0$, $n = 10000$

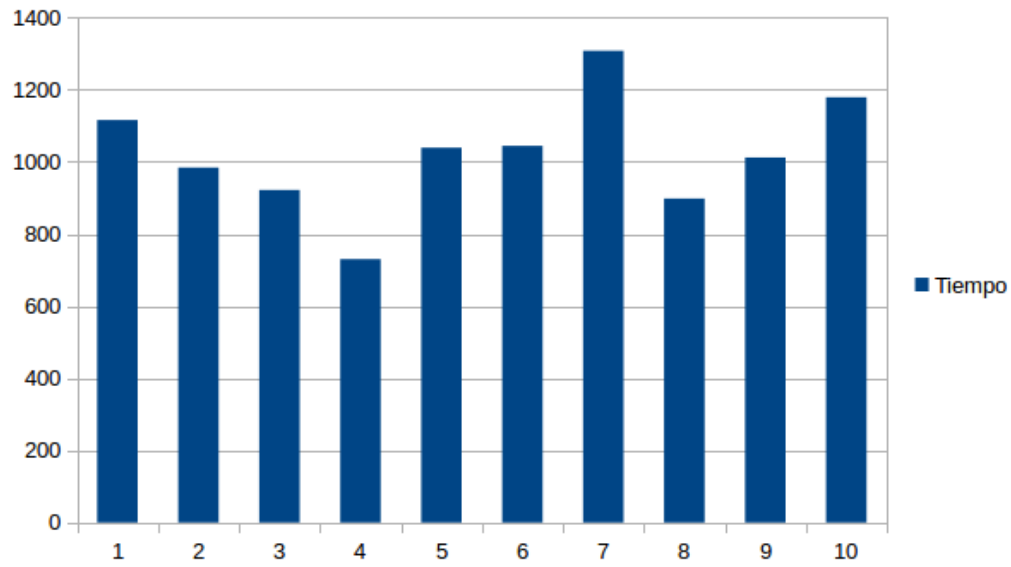


Figure 17: QuickSelect para $k = n/2$, $n = 10000$

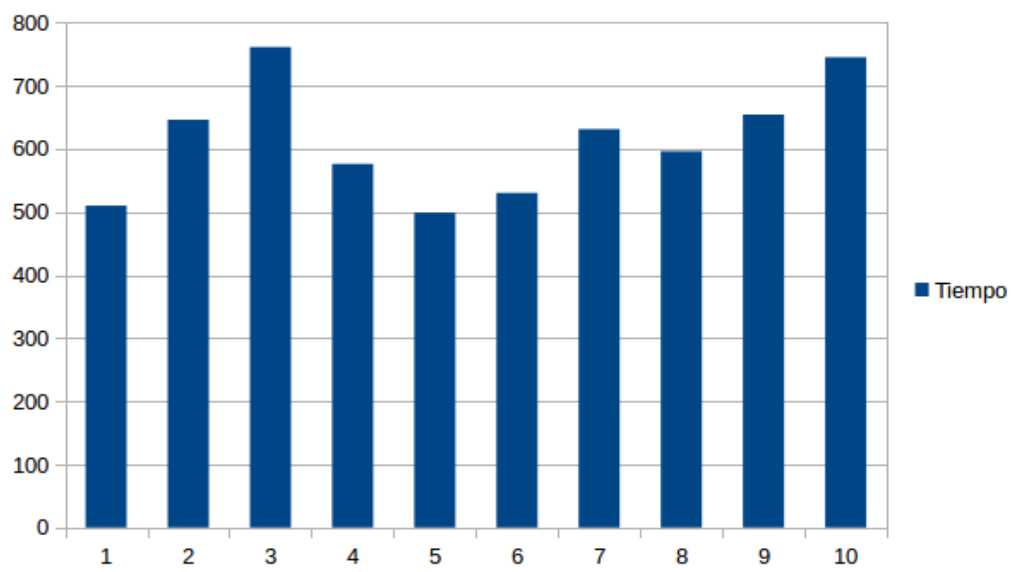


Figure 18: QuickSelect para $k = n-1$, $n = 10000$

1.7 Conclusiones

Para $k = 0$, tenemos:

- Fuerza Bruta: $O(n^2)$
- Ordenar y seleccionar: $O(n \log(n))$
- K-Select: $O(n)$
- K-HeapSort: $O(n)$
- HeapSelect: $O(n \log(n))$
- QuickSelect: $O(n)$

Las mejores elecciones serían:

- K-Select: Básicamente es el más intuitivo. La idea es recorrer el arreglo para buscar el mínimo. Esto es $O(n)$.
- K-HeapSort: Lo que hacemos es un heapify y extraer el mínimo. Esto es $O(n)$.
- QuickSort: Este algoritmo es $O(n)$, pero corremos el riesgo de que si se elige un mal pivote, pueda llegar a su peor caso que es $O(n^2)$.

La mejor elección en cuanto a tiempo es elegir K-Select.

Para $k = n/2$, tenemos:

- Fuerza Bruta: $O(n^2)$
- Ordenar y seleccionar: $O(n \log(n))$
- K-Select: $O(n^2)$
- K-HeapSort: $O(n \log(n))$
- HeapSelect: $O(n \log(n))$
- QuickSelect: $O(n)$

La mejor elección en cuanto a tiempo promedio es elegir QuickSelect.

Para $k = n-1$, tenemos:

- Fuerza Bruta: $O(n^2)$
- Ordenar y seleccionar: $O(n \log(n))$
- K-Select: $O(n^2)$
- K-HeapSort: $O(n \log(n))$
- HeapSelect: $O(n)$
- QuickSelect: $O(n)$

En este caso, las mejores elecciones serían HeapSelect y QuickSelect. El HeapSelect pasa a ser un heapify solamente. En cuanto a tiempos, son bastante parecidos. En los gráficos se puede ver que el QuickSelect alcanza un pico mayor que HeapSelect, debido a una mala elección de pivote.

2 Recorridos en grafos

2.1 BFS

2.1.1 Implementación

```
1 #include <fstream>
2 #include <sstream>
3 #include "../Headers/Digraph.h"
4
5 using namespace std;
6
7
8 Digraph::Digraph(int vertices) {
9     this->initGraph(vertices);
10 }
11
12 void Digraph::initGraph(int vertices) {
13     this->v = vertices;
14     this->e = 0;
15     //Reservo espacio para cada una de las listas de adyacencia ←
16     //de cada vertice
17     this->adjList.reserve(vertices);
18
19     for(int i=0; i<vertices;i++){
20         this->adjList.push_back(new std::list<Edge >);
21     }
22 }
23
24 int Digraph::getVertices(){
25     return this->v;
26 }
27
28 int Digraph::getEdges(){
29     return this->e;
30 }
31
32 list<Edge > Digraph::getAdjList(int v){
33     return this->adjList[v];
34 }
35
36 list<int> Digraph::getAdj(int v){
```

```

37     list<Edge > listVert = this->adjList[v];
38     list<int> vert;
39
40     //Devuelvo los vertices alcanzados por las aristas que salen↵
        de v
41     for (std::list<Edge >::iterator it=listVert->begin(); it != ↵
        listVert->end(); ++it){
42         vert.push_back(( it)->getDest());
43     }
44
45     return vert;
46 }
47
48 void Digraph::addEdge(int source, int dest, double weight){
49     Edge e = new Edge(source,dest,weight);
50     this->adjList[source]->push_back(e);
51     this->e ++;
52 }
53
54
55 Digraph::~Digraph() {
56     for(int i=0;i<this->v;i++){
57         list<Edge > list = this->adjList[i];
58         while(!list->empty()){
59             //Se elimina cada una de las aristas instanciadas
60             delete(list->front());
61             list->pop_front();
62         }
63         //Se elimina la lista de adyacencia para ese vertice
64         delete(list);
65     }
66 }
67
68
69 Digraph::Digraph(std::string fileName) {
70     std::ifstream file(fileName.c_str());
71     if (!file.is_open()) {
72         throw std::string("Error al abrir el archivo para ↵
            levantar el grafo") + fileName;
73     }
74
75     std::string line;
76
77     std::getline(file, line);
78     int vertexCount = atoi(line.c_str());

```

```

79
80     this->initGraph(vertexCount);
81
82     while (std::getline(file, line)) {
83         if (line.empty()) break;
84
85         std::istringstream stringstream(line);
86         std::string field;
87         getline(stringStream, field, ',');
88         int vertexFrom = atoi(field.c_str());
89         getline(stringStream, field, ',');
90         int vertexTo = atoi(field.c_str());
91         getline(stringStream, field, ',');
92         double weight = std::stod(field.c_str());
93
94         this->addEdge(vertexFrom, vertexTo, weight);
95     }
96
97     file.close();
98 }

```

```

1 #include "../Headers/Path.h"
2
3 Path::Path(Digraph g, int source, int dest){
4     this->g = g;
5     this->source = source;
6     this->dest = dest;
7     this->marked = new bool[g->getVertices()];
8     this->distance = new double[g->getVertices()];
9     this->edgeTo = new Edge [g->getVertices()];
10
11     for(int i=0; i < g->getVertices(); i++) {
12         this->marked[i] = false;
13         this->distance[i] = Path::NO_PATH;
14         this->edgeTo[i] = NULL;
15     }
16 }
17
18 Path::~~Path() {
19     delete marked;
20     delete edgeTo;
21     delete distance;
22 }
23

```

```

24 bool Path::visited(int v) {
25     return this->marked[v];
26 }
27
28 std::list<Edge > Path::pathTo(int v) {
29     std::list<Edge > aList;
30     if (!this->marked[v]) {
31         return aList; // Lista vacia
32     }
33
34     for(int i = v; i!= this->source; i=edgeTo[i]->getSource()) {
35         aList.push_back(edgeTo[i]);
36     }
37
38     aList.reverse();
39
40     return aList;
41 }
42
43 int Path::getSource() {
44     return this->source;
45 }
46
47 int Path::getDest() {
48     return this->dest;
49 }
50
51 std::list<int> Path::getVertexPath(int v) {
52     std::list<int> aList;
53     if (!this->marked[v]) {
54         return aList; // Lista vacia
55     }
56
57     aList.push_back(v);
58     for(int i = v; i!= this->source; i=this->edgeTo[i]->←
        getSource()) {
59         aList.push_back(this->edgeTo[i]->getSource());
60     }
61
62     aList.reverse();
63
64     return aList;
65 }
66
67 double Path::distanceTo(int v) {

```

```

68     return this->distance[v];
69 }

```

```

1  #include "../Headers/PathBFS.h"
2
3  #include <queue>
4
5  using namespace std;
6
7  PathBFS::PathBFS(Digraph g, int source, int dest) : Path(g, ←
    source, dest) {
8
9      for(int i=0;i<g->getVertices();i++) {
10         marked[i] = false;
11         distance[i] = Path::NO_PATH; //Inicializo ←
            distancias con distancia infinito
12     }
13
14     queue<int> queue;
15
16     marked[this->source] = true;
17     distance[this->source] = 0;
18
19     queue.push(this->source); //arranco desde el source
20
21     //Mientras haya un vertice pendiente y no se haya encontrado←
        el destino
22     while(!queue.empty() && !marked[this->dest]){
23         int v = queue.front();
24         queue.pop();
25         list<Edge> adjList = g->getAdjList(v);
26
27         //Barro la lista de adyacencia
28         for (std::list<Edge>::iterator it=adjList->begin(); it ←
            != adjList->end(); ++it){
29             int vert = (it)->getDest();
30             double weight = (it)->getWeight();
31
32             if(!marked[vert]){
33                 distance[vert] = distance[v] + weight; // ←
                    Acumulo la distancia desde el origen hasta ←
                    el vertice
34                 marked[vert]= true;
35                 queue.push(vert);

```

```

36         edgeTo[vert] = it; //guardo el camino por el ←
           que fui
37     }
38 }
39 }
40 }

```

2.1.2 Complejidad

El algoritmo BFS es un algoritmo Greedy que explora iteración tras iteración los nodos adyacentes, que no hayan sido visitados, de los nodos ya visitados, comenzando por un nodo particular al que se conoce como **origen**.

De esta forma para encontrar el camino hacia un nodo particular, el **destino**, el algoritmo va formando un conjunto de nodos visitados que se expande desde el origen y finaliza cuando se encuentra el nodo deseado o cuando recorre todos los nodos y no lo encuentra. En este ultimo caso, dado que ya no quedan nodos en la componente conexa que contiene el origen, se puede decir que no existe un camino hacia el nodo destino.

El algoritmo tiene complejidad $O(m + n)$, siendo m la cantidad de aristas y n la cantidad de vértices.

Sea n_u el grado del nodo u , el número de aristas incidentes a u . En tiempo insumido para recorrer en una iteración del while las aristas incidentes al nodo u es $O(n_u)$. Por lo tanto, el total sobre todos los nodos es $O(\sum_{u \in V} n_u = 2m) = O(m)$.

El tiempo insumido en considerar las aristas sobre todo el algoritmo es $O(m)$.

Luego tenemos $O(n)$ para lo que es la inicialización de las estructuras.

2.1.3 Mejor y peor caso

El peor caso se tiene cuando el origen y el destino tienen distancia máxima, entendiéndose por tal la mayor separación posible de nodos. O bien, cuando no hay camino ya que el algoritmo debe recorrer todo el grafo hasta determinarlo.

En particular para este algoritmo, es difícil encontrar un mejor caso. Es útil cuando el grafo es denso y no hay mucha distancia entre los nodos origen y destino. En este punto, cabe considerar que mientras mas alejado estén los nodos, peor la performance.

Solo es útil para grafos que posean igual peso en las aristas, en este caso, el camino que encuentra es mínimo. Cuando se agrega pesos diferentes en las aristas ya no asegura que el camino encontrado sea mínimo.

2.1.4 Gráficos

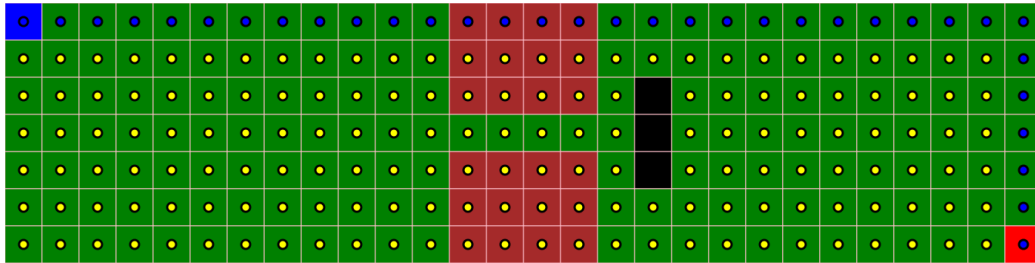


Figure 19: Recorrido BFS con máxima distancia

En el gráfico se puede observar varios puntos interesantes del algoritmo BFS.

Primero notar que el grafo sobre el que se aplicó el algoritmo son nodos con cuatro aristas que los conectan con los nodos inferior, superior, izquierdo y derecho. El nodo azul es el origen y el rojo el destino. Los nodos en verde tienen peso 1 y los marrones 10. Los nodos en negro son aislados. Los puntos marcados en oscuro significan que el nodo es parte del camino hacia el destino. Los nodos claro que el algoritmo visitó el nodo en la búsqueda del camino.

Habiendo aclarado esto se puede observar que el algoritmo recorrió todos los nodos para encontrar el camino, esto se condice con que los nodos están a máxima distancia. Fue necesario recorrer todo el grafo.

Si no contemplamos los pesos el camino es mínimo. Sin embargo el camino no es mínimo para nodos pesados. Esto es evidente si notamos que el camino mínimo deberá pasar por los nodos verdes entre los marrones.

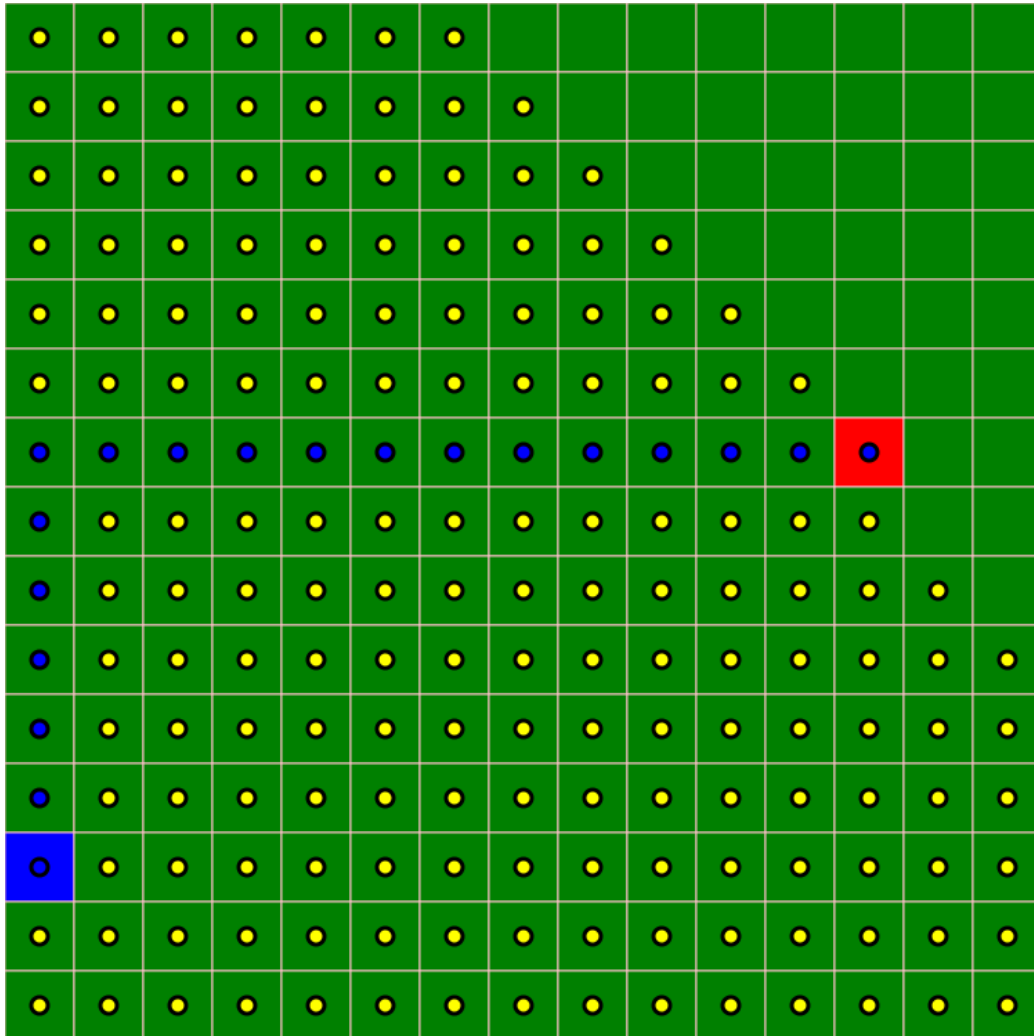


Figure 20: Recorrido BFS con media distancia

En este caso se puede observar como se comporta el algoritmo cuando la distancia comienza a reducirse, dado que ya no requiere recorrer todos los nodos (no recorre aquellos con distancia mayor a la distancia con el nodo destino). En este caso el camino encontrado es minimo.

2.2 Dijkstra

2.2.1 Implementación

```
1 Dijkstra::Dijkstra(Digraph g, int source, int dest) : Path(g, source, dest) {
2
3     for(int i=0; i < g->getVertices(); i++) {
4         this->marked[i] = false;
5         this->distance[i] = Path::NO_PATH;           // Inicializo
6                                                     distancias con distancia infinito
7     }
8     this->marked[this->source] = true;
9     this->distance[this->source] = 0;
10
11     PriorityQueue priorityQueue;
12     priorityQueue.push(this->source, 0 );
13
14     while (!priorityQueue.empty()) {
15         std::pair<int, double> pairObject = priorityQueue.pop();
16         int currentVertex = pairObject.first;
17         double priority = pairObject.second;
18
19         if (currentVertex == this->dest) {
20             break;
21         }
22
23         std::list<Edge > adjList = g->getAdjList(currentVertex);
24
25         for (std::list<Edge >::iterator it=adjList->begin(); it != adjList->end(); ++it){
26             int nextVert = ( it)->getDest();
27             double weight = ( it)->getWeight();
28
29             double newCost = this->distance[currentVertex] + weight;
30
31             if (!this->marked[nextVert] || newCost < this->distance[nextVert]) {
32                 this->marked[nextVert] = true;
33                 this->distance[nextVert] = newCost;
34                 priority = newCost;
```

```

35         priorityQueue.push(nextVert, priority);
36         edgeTo[nextVert] = it;
37     }
38 }
39 }
40 }

```

2.2.2 Complejidad

Antes de comenzar a analizar el algoritmo, necesitamos saber cuanto cuestan las operaciones de la cola de prioridad. Tenemos tres operaciones que usamos: push, pop, empty. Las operaciones push y top son $O(\log(n))$, siendo n la cantidad de elementos en la cola [7][8]. La operación empty es $O(1)$ [6].

Inicialmente, tenemos todo lo que es la inicialización de las estructuras para almacenar la información. Toda esta parte es $O(n)$, siendo n la cantidad de vértices.

La máxima cantidad de cosas que podemos insertar en la cola de prioridad está dado por el número de aristas, m . Por lo que, en el peor caso, iteraremos m veces.

Dentro de cada iteración, realizamos una operación pop para extraer el elemento de mayor prioridad y una operación push por cada arista del vértice actual siempre que mejoremos la distancia hacia el otro vértice incidente o dicho vértice incidente no haya sido marcado. Realizar la verificación de si se mejora la distancia o no podría omitirse, ya que en el peor de los casos haríamos m push sobre la cola de prioridad. La operación de pop y push son $O(\log(m))$. En el peor de los casos $m = n^2$, por lo que nos quedaría que la operación pop y push son $O(\log(m)) = O(\log(n^2)) = O(2\log(n)) = O(\log(n))$.

En conclusión, Dijkstra tiene una complejidad de $O(n + m\log(n))$, siendo n la cantidad de vértices y m la cantidad de aristas.

2.2.3 Mejor y peor caso

Peor Caso

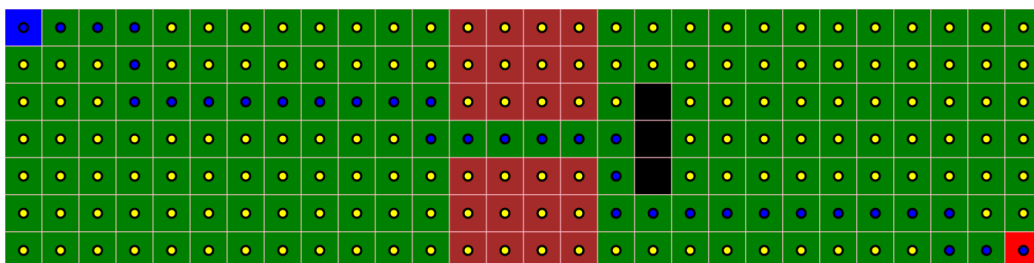


Figure 21: Dijkstra peor caso

Como podemos ver en la figura de arriba, mientras más alejado el vértice destino esté del origen, mayor será el tiempo insumido por el algoritmo. Los puntos amarillos son todos los vértices procesados, en este caso todos los del grafo. Podemos ver que Dijkstra evitó pasar por los nodos de color rojo, que son los más pesados.

Mejor caso

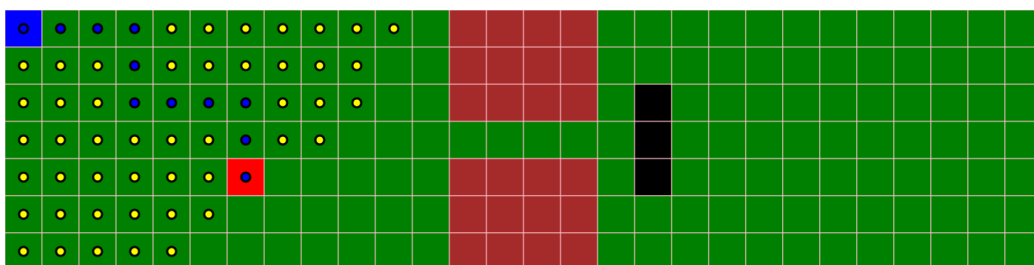


Figure 22: Dijkstra mejor caso

Como podemos ver en la figura de arriba, se puede apreciar que el algoritmo de Dijkstra no tuvo la necesidad de visitar todos los nodos. Tan pronto encuentre el camino de menor distancia finaliza.

2.2.4 Análisis

Dijkstra tiene la ventaja de que siempre nos va a dar el camino óptimo para llegar desde un origen hasta un destino dado. Una de las desventajas es la cantidad de operaciones que requiere. Dependiendo de la circunstancia, por ejemplo en un juego que tenga un mapa y el jugador necesite ir de un extremo a otro, podría ser una mejor elección un algoritmo tipo "Búsqueda con heurística" para tratar de visitar la menor cantidad de nodos posibles.

Si el grafo no tiene pesos, la mejor opción es descartar Dijkstra y utilizar BFS.

2.3 Búsqueda con heurística

2.3.1 Implementación

```
1
2 #include "../Headers/SearchWithHeuristic.h"
3 #include "../Headers/PriorityQueue.h"
4
5 SearchWithHeuristic::SearchWithHeuristic(Digraph g, int source↵
6     , int dest, Heuristic& heuristic) : Path(g, source, dest) {
7
8     for(int i=0; i < g->getVertices(); i++) {
9         marked[i] = false;
10        distance[i] = Path::NO_PATH; // Inicializo ↵
11        distancias con distancia infinito
12    }
13
14    PriorityQueue priorityQueue;
15
16    marked[this->source] = true;
17    distance[this->source] = 0;
18
19    priorityQueue.push(this->source, 0); // Arranco desde el ↵
20    source.
21
22    //Mientras haya un vertice pendiente y no se haya ↵
23    encontrado el destino
24    while(!priorityQueue.empty() && !marked[this->dest]){
25        std::pair<int, double> pairObject = priorityQueue.pop();
26        int v = pairObject.first;
27
28        std::list<Edge > adjList = g->getAdjList(v);
29
30        //Barro la lista de adyacencia
31        for (std::list<Edge >::iterator it=adjList->begin(); it↵
32            != adjList->end(); ++it){
33            int vert = ( it)->getDest();
34            double weight = ( it)->getWeight();
35
36            if(!marked[vert]){
37                distance[vert] = distance[v] + weight; // ↵
38                Acumulo la distancia desde el origen hasta ↵
39                el vertice
40                marked[vert]= true;
41            }
42        }
43    }
```

```

34         priorityQueue.push(vert, heuristic.getCost(vert↔
           ));
35         edgeTo[vert] = it; //guardo el camino por el ↔
           que fui
36     }
37 }
38 }
39 }

```

2.3.2 Complejidad

Tenemos una inicialización $O(n)$. Luego, tenemos un ciclo para recorrer la cola de prioridad hasta que esté vacía. En este algoritmo tenemos una ligera diferencia. Vamos a insertar siempre que el vértice no haya sido visitado. Por lo tanto, como mucho, la cantidad de elementos que podría llegar a sacar de la cola de prioridad sería n , siendo n la cantidad de vértices.

Dentro de cada iteración, lo que hacemos es una operación pop para extraer el elemento de mayor prioridad y una operación push por cada arista del vértice actual para las cuales el otro vértice incidente no haya sido marcado. Es decir, en el peor de los casos, cada arista va a provocar que se realice un push en la cola de prioridad.

Complejidad del algoritmo: $O(n + m \log(n))$

2.3.3 Mejor y peor caso

Mejor caso

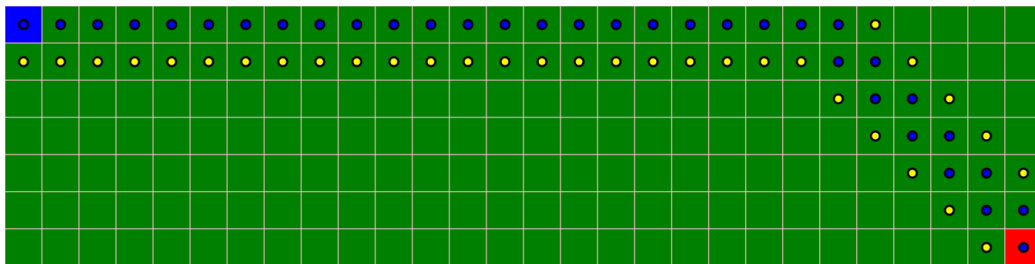


Figure 23: Búsqueda con heurística, utilizando una heurística euclidiana en un caso ideal

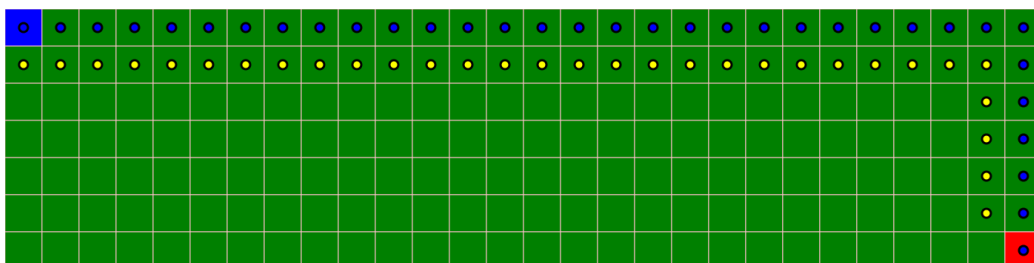


Figure 24: Búsqueda con heurística, utilizando una heurística Manhattan en un caso ideal

Como se puede ver, en un caso ideal, donde todas las aristas tienen igual peso, ambas heurísticas dan caminos óptimos. Pero se puede pensar un caso sencillo para complicar a estas heurísticas.

Peor caso

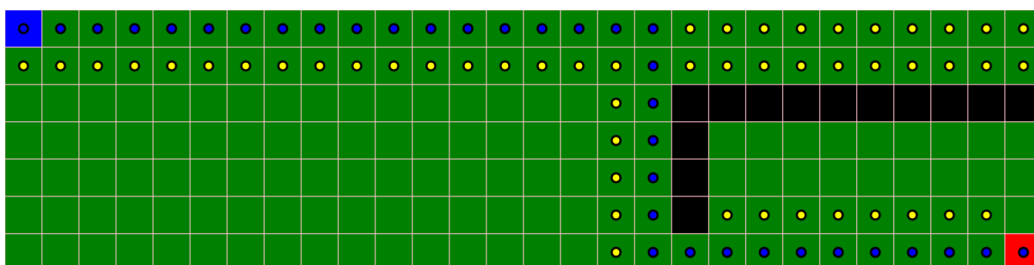


Figure 25: Búsqueda con heurística, utilizando una heurística euclidiana en un caso no tan ideal

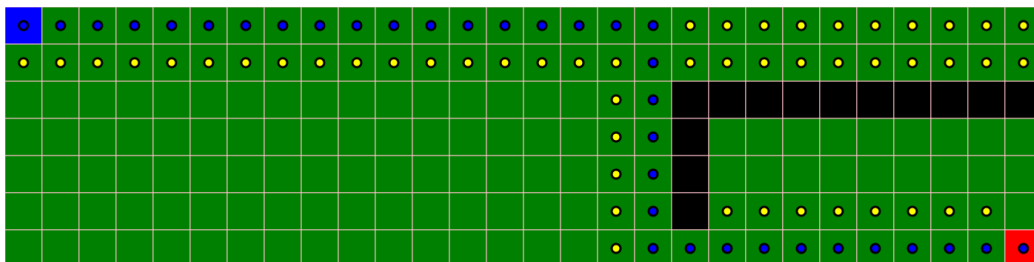


Figure 26: Búsqueda con heurística, utilizando una heurística Manhattan en un caso no tan ideal

Podemos ver que colocando un obstáculo, provoca que el algoritmo tenga que analizar más vértices.

2.3.4 Análisis

Heurística exacta

Veamos que solución obtenemos si utilizamos una heurística exacta que nos diga el costo exacto desde un vértice dado hasta el destino.

```

1 #include "../Headers/ExactHeuristic.h"
2 #include "../Headers/Dijkstra.h"
3
4 ExactHeuristic::ExactHeuristic(Digraph digraph, int dest) {
5     this->digraph = digraph;
6     this->dest = dest;
7 }
8
9 double ExactHeuristic::getCost(int v) {
10     Dijkstra dijkstra(this->digraph, v, this->dest);
11     return dijkstra.distanceTo(this->dest);
12 }

```

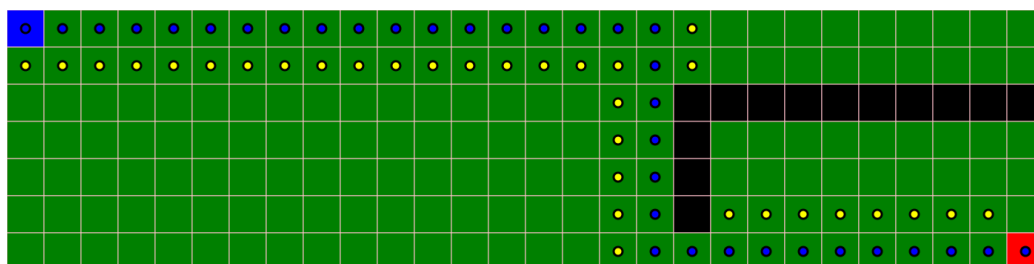


Figure 27: Búsqueda con heurística, utilizando una heurística exacta

Podemos ver que utilizando la heurística exacta, el algoritmo evitó visitar los vértices de la esquina superior derecha, evitando de esta forma el obstáculo.

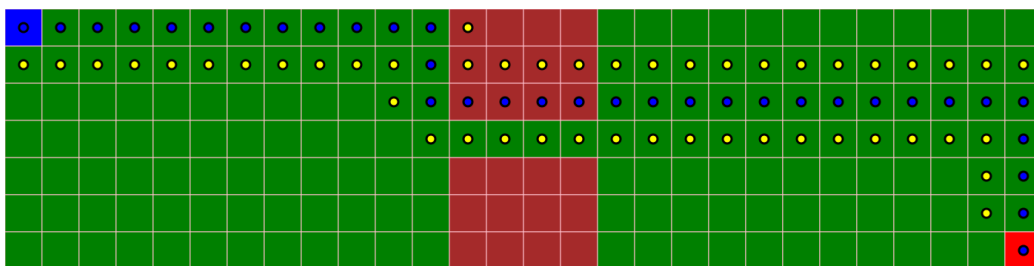


Figure 28: Búsqueda con heurística, utilizando una heurística exacta

Utilizar una heurística exacta que nos diga el costo exacto desde un vértice dado hasta el destino no nos garantiza que obtengamos un camino óptimo, como se puede ver en el ejemplo.

Heurística pesimista

La heurística que probaremos será una que utilice BFS para obtener el camino desde un vértice v hasta el vértice destino. Obviamente, que BFS no toma el camino óptimo siempre, por lo que algunas veces nos dará distancias no óptimas.

```

1 #include "../Headers/PessimisticHeuristic.h"
2 #include "../Headers/Dijkstra.h"
3 #include "../Headers/PathBFS.h"
4
5 PessimisticHeuristic::PessimisticHeuristic(Digraph digraph, ↵
6     int dest) {
7     this->digraph = digraph;
8     this->dest = dest;
9 }
10
11 double PessimisticHeuristic::getCost(int v) {
12     PathBFS pathBFS(this->digraph, v, this->dest);
13     return pathBFS.distanceTo(this->dest);
14 }

```

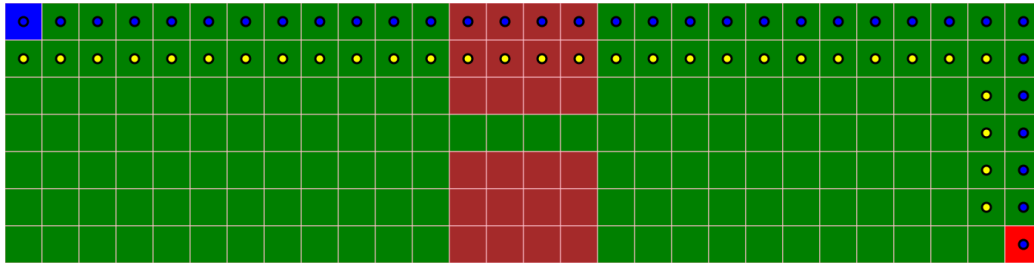


Figure 29: Búsqueda con heurística, utilizando una heurística pesimista

Como se puede ver, no obtenemos un camino óptimo.

Heurística optimista

Las heurísticas optimistas que probamos son las heurísticas Manhattan y Euclidean, que se puede ver en el apartado de "Mejor y peor caso".

2.4 A*

2.4.1 Implementación

```
1 AStar::AStar(Digraph g, int source, int dest, Heuristic &heuristic) : Path(g, source, dest) {
2
3     for(int i=0; i < g->getVertices(); i++) {
4         this->marked[i] = false;
5         this->distance[i] = Path::NO_PATH;
6         this->edgeTo[i] = NULL;
7     }
8
9     this->marked[this->source] = true;
10    this->distance[this->source] = 0;
11
12    PriorityQueue priorityQueue;
13    priorityQueue.push(this->source, 0 );
14
15    while (!priorityQueue.empty()) {
16        std::pair<int, double> pairObject = priorityQueue.pop();
17        int currentVertex = pairObject.first;
18        double priority = pairObject.second;
19
20        if (currentVertex == this->dest) {
21            break;
22        }
23
24        std::list<Edge > adjList = g->getAdjList(currentVertex);
25
26        for (std::list<Edge >::iterator it=adjList->begin(); it != adjList->end(); ++it){
27            int nextVert = ( it)->getDest();
28            double weight = ( it)->getWeight();
29
30            double newCost = this->distance[currentVertex] + weight;
31
32            if (!this->marked[nextVert] || newCost < this->distance[nextVert]) {
33                this->marked[nextVert] = true;
34                this->distance[nextVert] = newCost;
```

```

35         priority = newCost + heuristic.getCost(nextVert↵
36             );
37         priorityQueue.push(nextVert, priority);
38         edgeTo[nextVert] = it;
39     }
40 }
41 }

```

2.4.2 Complejidad

La complejidad es igual a la de Dijkstra ya que lo único que cambia es el valor de la prioridad con la cual se insertan los vértices en la cola de prioridad.

Complejidad del algoritmo: $O(n + m \log(n))$.

2.4.3 Mejor y peor caso

El mejor y peor caso lo discutimos en general en la sección de análisis.

2.4.4 Análisis

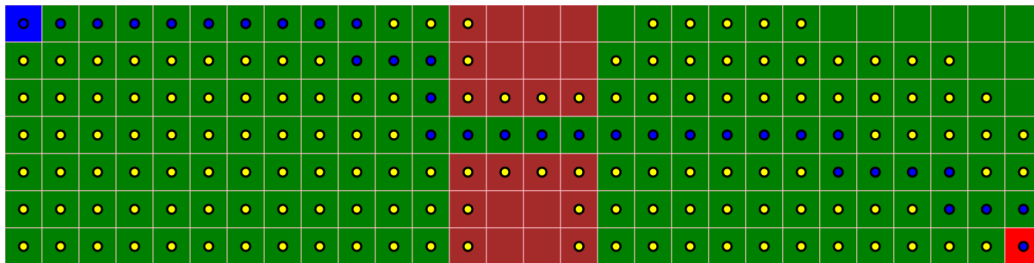


Figure 30: AStar,utilizando heurística euclidiana

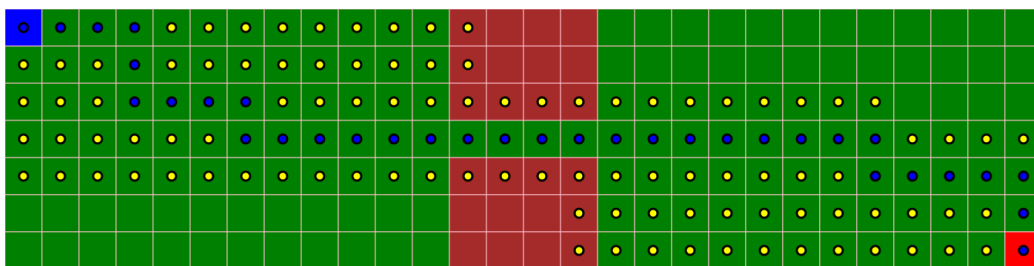


Figure 31: AStar,utilizando heurística exacta

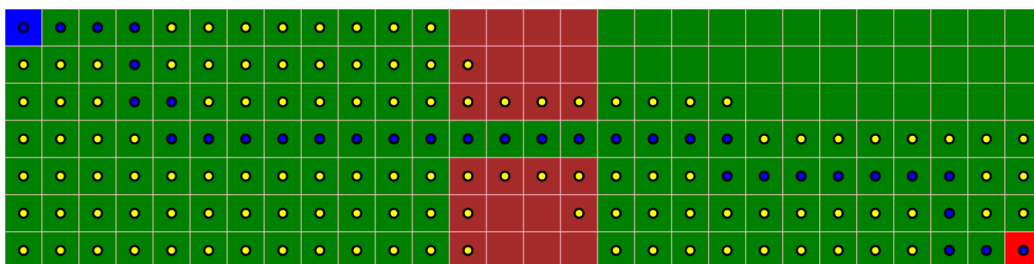


Figure 32: AStar,utilizando heurística Manhattan



Figure 33: AStar,utilizando heurística pesimista

Con este ejemplo, en todos los casos obtuvimos la solución óptima. En el caso de la heurística Manhattan y la heurística Euclidean, vemos que hay una gran cantidad de vértices analizados. Esto se debe principalmente a que hay más de un camino con la misma longitud. Para atacar este problema, se podría aplicar alguna estrategia para decidir en caso de empate [10]. Por supuesto, mientras más “optimista” sea la heurística, esto es, mientras menor sea el valor que devuelva $h(n)$, el algoritmo de A se irá pareciendo más a Dijkstra y con ellos traerá los problemas que tiene Dijkstra.

Si utilizamos una heurística exacta, esto es que devuelva el costo exacto entre un vértice v y el vértice destino, A^* va a seguir el mejor camino posible, expandiéndose lo menos posible (en este caso, analiza vértices de más debido al existir varios caminos con igual costo) [10].

Se puede observar que tuvimos un poco de suerte con la heurística pesimista, ya que fue la que menor trabajo realizó. Si hacemos que la heurística sea demasiado pesimista, esto es, $h(n)$ muy grande en comparación a $d(n)$, transformaremos A^* en el algoritmo de Búsqueda con heurística. Es por esto que es importante respetar las escalas entre la función de heurística y los costos de las aristas.

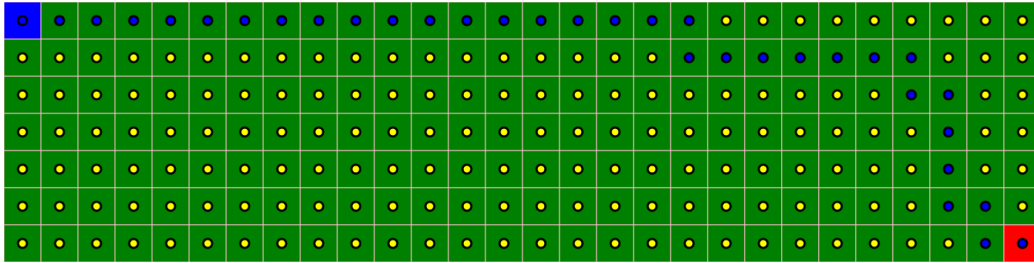


Figure 34: AStar,utilizando heurística Euclidean

En la figura 34 se puede observar que al haber tantos caminos de igual costo, provoca que se visiten todos los nodos.

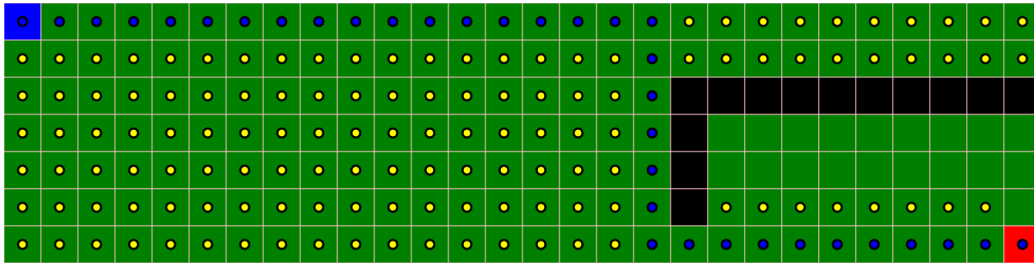


Figure 35: AStar,utilizando heurística Euclidean

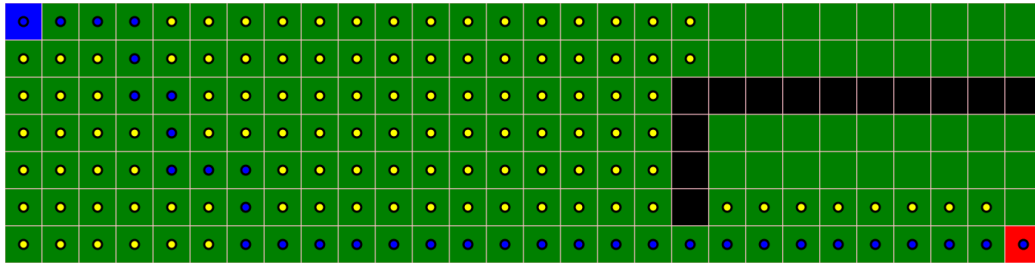


Figure 36: AStar,utilizando heurística exacta

Con la heurística exacta, recorrimos menor cantidad de vértices en comparación con la heurística Euclidean.

Referencias

- [1] `std::sort`
<http://en.cppreference.com/w/cpp/algorithm/sort>
- [2] `std::make_heap`
http://en.cppreference.com/w/cpp/algorithm/make_heap
- [3] `std::pop_heap`
http://en.cppreference.com/w/cpp/algorithm/pop_heap
- [4] `std::push_heap`
http://en.cppreference.com/w/cpp/algorithm/push_heap
- [5] Introduction to Algorithms, Third Edition, Cormen; Capítulo 9
- [6] `std::priority_queue::empty`
http://en.cppreference.com/w/cpp/container/priority_queue/empty
- [7] `std::priority_queue::pop`
http://en.cppreference.com/w/cpp/container/priority_queue/pop
- [8] `std::priority_queue::push`
http://en.cppreference.com/w/cpp/container/priority_queue/push
- [9] QuickSelect
<https://en.wikipedia.org/wiki/Quickselect>
- [10] Heurísticas
<http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html>
breaking-ties
<http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html>
a-stars-use-of-the-heuristic