Algorítmica

Curso 2023-2024

Grupo Viterbi

PRÁCTICA 3-ALGORITMOS GREEDY

Integrantes:

Miguel Ángel De la Vega Rodríguez	miguevrod@correo.ugr.es
Alberto De la Vera Sánchez	joaquin724@correo.ugr.es
Joaquín Avilés De la Fuente	adelaveras01@correo.ugr.es
Manuel Gomez Rubio	e.manuelgmez@go.ugr.es
Pablo Linari Perez	e.pablolinari@go.ugr.es

Facultad de Ciencias UGR
Escuela Técnica Ingeniería Informática UGR
Granada
2023-2024

Índice general

Apartado 1

Autores

- **Miguel Ángel De la Vega Rodríguez:** 20%
  - Algoritmos del Viajante
  - Redacción memoria sección Viajante
- **Joaquín Avilés De la Fuente:** 20%
  - Programacion SumaMax (DyV)
  - Estudio eficiencia teórica algoritmos específicos y DyV
  - Tests de eficiencia
- **Alberto De la Vera Sánchez:** 20%
  - Redacción LaTeX
  - Estudio de umbrales teóricos y empíricos de DyV
  - Graficas y ajustes
- **Manuel Gomez Rubio** 20%
  - Programacion SumaMax (Kadane)
  - Programacion Losetas
- **Pablo Linari Pérez:** 20%
  - Programacion SumaMax (Kadane)
  - Programacion Losetas

Apartado 2

Equipo de trabajo

- **Miguel Ángel De la Vega Rodríguez:** (Ordenador donde se ha realizado el computo)
  - AMD Ryzen 7 2700X 8-Core
  - 16 GB RAM DDR4 3200 MHz
  - NVIDIA GeForce GTX 1660 Ti
  - 1 TB SSD NvMe
  - Debian 12 Bookworm
  - Compilador GCC 12.2.0

Apartado 3

Asignación de aulas

En esta sección se estudia lo referente al algoritmo usado para resolver el segundo problema el cual nos pide lo siguiente :

Se quieren ralizar N exámenes en un día en la ETSIIT , el centro cuenta con m aulas donde m es mayor que N . Sin embargo cada vez que se utiliza un aula el centro debe contratar un vigilante por tanto se debe diseñar un algoritmo que clacule en base al horario de los exámenes el menor costo posible para la escuela , es decir hay que encontrar la manera de distribuir los exámenes para usar el menor número de aulas posibles .

3.1 Estructura de los datos

Una vez planteado el problema porcedemos a describir las herramientas que hemos usado para resolverlo Para representar los horarios de los exámenes hemos usado la siguiente estructura :

```
1   struct tiempo{
2       int hora;
3       int min;
4       tiempo(int h , int m){
5           hora = h;
6           min = m ;
7       }
8   };
9   struct examen{
10      tiempo inicio; // Hora de inicio
11      tiempo final; // Hora de finalizaci n
12      examen(tiempo t , tiempo t1) : inicio(t), final(t1){}
13  };
14
15  struct compare{
16      bool operator()(const examen&a,const examen&b){
17          if(a.inicio.hora==b.inicio.hora){
18              return a.inicio.min < b.inicio.min;
19          }
20          return a.inicio.hora < b.inicio.hora;
21      }
22  };
```

El struct compare se utiliza para ordenar el vector de entrada de exámenes de manera que se ordenen de menor a mayor en función de la hora de inicio de los exámenes , de esta manera se facilita la ejecución del algoritmo que se encarga de asignar los exámenes a las aulas de la manera más eficiente posible.

3.2 Algoritmo

A continuación vemos el algoritmo usado para la distribución de las aulas.

```
bool sesolapan(examen e1,examen e2){
      if(e1.final.hora == e2.inicio.hora){
          return e1.final.min >e2.inicio.min;
      }
      return e1.final.hora > e2.inicio.hora;
  }

  int minAulas(const vector<examen> & v){

      // Conjunto de candidatos v

      int naulas = 1;
      vector<queue<examen>> horario;
      //Conjunto de seleccionados horario (vector de colas , cada
          cola es un aula)

      horario.emplace_back(queue<examen>());
      horario[0].push(v[0]);
      int i;
      //
      for(int nex = 1;nex < v.size();nex++){
          i=0;
          //Funcion de seleccion
          while(i<horario.size() && (sesolapan(horario[i].back(),
              v[nex]))){i++;}
          //Funcion de factibilidad
          if(i==horario.size()){
              horario.emplace_back(queue<examen>());
              horario[i].push(v[nex]);
              naulas++;
          }
          else{
              horario[i].push(v[nex]);
          }
      }
      return naulas; //Funcion objetivo
}
```

Uno de los factores importantes que hay que destacar de este algoritmo es que el vector **v** se supone ordenado de mayor a menor , es decir que en el vector los exámenes aparecen en orden creciente de horario de inicio de esta manera la asignación de aulas es más rápida y eficiente.

El algoritmo comienza añadiendo el primer examen al primer aula , cada aula se representa con una posición del vector y el horarario del aula se representa con una cola ya que solo nos interesa el último examen hecho en el aula.

Posteriormente se entra al bucle for el cual recorre todo el vector de candidatos ya que todos los exámenes deben ser realizados .

Dentro del bucle for se entra a un while el cual comprueba mediante la función **sesolapan(examen e1 ,examen e2)** si el examen que está buscando un aula se solapa con el que se está realizando en el aula , si no se solapan se añade a ese aula , si se solapan se busca otra aula en la que no se solapen y se añade a esa aula. En el caso de que se hallan agotado las aulas actuales se crea una nueva aula y se añade el examen a esa aula.

3.3 Justificación Greedy

En este apartado vamos a analizar el algoritmo identificando los componentes que lo hacen un algoritmo greedy y demostrando que siempre llega a la solución óptima.

Apartado 4

Viajante

En esta sección se analiza todo aquello referente al cuarto problema propuesto, el problema del viajante. Siguiendo las directrices indicadas, se han estudiado y diseñado diferentes algoritmos "Greedy"que aproximan el problema, con el objetivo final de comparar cual nos proporciona mejores resultados en cuanto a términos de eficiencia y precisión. Para determinar esto último, también cabe destacar detalles de menor importancia como la complejidad de los algoritmos, la estabilidad frente a distintas entradas o la posibilidad de mejora de los mismos mediante la elección de distintos puntos de partida o con mejoras posteriores como pueden ser aquellas que proporcionan algoritmos como λ-opt o genéticos.

De aqui en adelante, describiremos por secciones los algoritmos implementados y al final proporcionaremos una sección comparativa sobre la cual nos basaremos para determinar conclusiones, en particular, elegiremos el algoritmo cuya solución consideremos más conveniente. En lo que sigue se muestra únicamente el resultado obtenido por el algoritmo sin mejoras posteriores, este tipo de mejoras suponen una pequeña desvirtuación del objetivo de encontrar el mejor algoritmo y es por ello, que su uso se limita a la conclusión final.

4.1 Algoritmo *Nearest Neighbour*

Tal y como el nombre indica, el primer algoritmo implementado no es nada más, ni nada menos que el primer algoritmo que probablemente se le puede ocurrir a cualquier persona que se enfrente a este problema. La idea es sencilla e intuitiva, nos dan un conjunto de puntos y queremos encontrar el camino más corto que los recorra, para ello, **elegimos** un punto inicial y a partir de ese punto, en un proceso iterativo, elegimos en cada paso el siguiente punto más cercano al actual, o lo que es lo mismo, el punto para el cual la distancia al punto actual es la menor (No confundir con el punto para el cual la distancia total es la menor, ya que, aunque parecido, no es lo mismo). Este proceso se repite hasta que todos los puntos han sido visitados.

A continuación se proporciona la implementación propuesta del algoritmo en cuestión:

```
vector<Point> nearestNeighborTSP(const vector<Point>& points) {
  vector<Point> path;
  vector<bool> visited(points.size(), false);
  path.reserve(points.size());
  path.emplace_back(points[0]);
  visited[0] = true;

  for (int i = 0; i < points.size() - 1; ++i) {
    double minDistance = numeric_limits<double>::max();
    int nearestNeighbor = -1;
```

```
    for (int j = 0; j < points.size(); ++j) {
      if (!visited[j]) {
        double distance = path[i].distanceTo(points[j]);
        if (distance < minDistance) {
          minDistance = distance;
          nearestNeighbor = j;
        }
      }
    }
    path.emplace_back(points[nearestNeighbor]);
    visited[nearestNeighbor] = true;
  }

  return path;
}
```

Como se puede apreciar, se ha hecho uso de un vector de puntos visitados (Programación dinámica) para evitar visitar un punto más de una vez, esto no supone una complejidad notable, sin embargo, si que supone una gran mejora en términos de eficiencia.

4.2 Algoritmo *Ordenación*

Como el nombre indica, este algoritmo consiste en ordenar los puntos de partida de acuerdo a un criterio especifico (en nuestro caso los hemos ordenado por la coordenada x de menor a mayor, teniendo en cuenta también para puntos cercanos en x, la coordenada y). Una vez ordenados los puntos, simplemente recorremos el vector de puntos en el orden en el que se encuentren, de esta forma, el camino total trata de reducir cruces y distancias con la idea intuitiva de que si dos puntos están cerca en el plano, probablemente, la distancia que se recorra al pasar por ellos, sea mínima. Como veremos en el siguiente algoritmo y en la conclusión, esto en la práctica no es del todo cierto. Primero veamos la implementación del algoritmo:

```
vector<Point> orderedTSP(const vector<Point>& points) {
      vector<Point> tour;
      tour.reserve(points.size());
      copy(points.begin(), points.end(), back_inserter(tour));
      sort(tour.begin(), tour.end());

      return tour;
}
```

Como se puede apreciar, la implementación es muy sencilla, simplemente se copian los puntos de partida a un vector auxiliar, se ordenan y se devuelve el vector ordenado, sin embargo cuando vemos el resultado de la ejecución, nos damos cuenta de que aunque los puntos esten muy cerca respecto a la coordenada x, los saltos que se dan en la coordenada y pueden ser muy grandes, lo que hace que el camino total sea mucho mayor de lo esperado, además para cerrar el camino, la distancia es la máxima posible en cuanto a x. Para mejorar esto, podemos observar que si nuestro problema son los saltos en la coordenada y y la distancia de cierre, podemos intentar minimizarlos, de donde surge el siguiente algoritmo.

4.3 Algoritmo *Circular*

Para solucionar el problema anterior, se propone una alternativa que se basa en la misma idea de ordenar por la coordenada x, sin embargo, en este caso la coordenada y solo se tiene en cuenta tras ordenar, lo que se hace es guardar el la coordenada y de los puntos máximo y mínimo y se calcula el punto medio entre ambos, esto se

hace para recorrer los puntos de forma circular como el nombre indica, de modo que los puntos que se quedan en la parte de abajo del plano, se recorren en sentido horario y los puntos que se quedan en la parte de arriba del plano, se recorren en sentido antihorario. De esta forma, se consigue minimizar la cantidad de saltos en el eje y y la distancia de cierre, resulta sugerente pensar que este proceso de división en dos partes, se puede hacer de forma recursiva, sin embargo, en la práctica, esto supone una complejidad superior a la hora de implementar mas bucles, y no supone una mejora significativa en términos de precisión, por lo que se ha optado por dejarlo de esta forma.

El problema que presenta esta implementación es que dependiendo de la forma que tenga el conjunto de puntos que se le pase, se obtendrán resultados muy distintos, sin embargo esto ocurre en todos los algoritmos, por lo que no se considera un problema en si mismo, ya que es inherente al problema. A continuación se muestra la implementación del algoritmo:

```
vector<Point> CircTSP(const vector<Point>& points) {
    vector<Point> tour;
    vector<Point> circularTour;
    tour.reserve(points.size());
    circularTour.reserve(points.size());
    tour = points;
    sort(tour.begin(), tour.end());

    int min = numeric_limits<int>::max();
    int max = numeric_limits<int>::min();

    for (int i = 0; i < points.size(); ++i) {
        min = min(min, points[i].getY());
        max = max(max, points[i].getY());
    }

    double mid = (max + min )/2;

    for (int i = 0; i < points.size(); ++i) {
        if (tour[i].getY() <= mid) {
            circularTour.emplace_back(tour[i]);
        }
    }

    for (int i = points.size(); i >= 0; --i) {
        if (tour[i].getY() > mid) {
            circularTour.emplace_back(tour[i]);
        }
    }

    return circularTour;
}
```

Una vez mas, podemos apreciar que se puede aumentar la eficiencia del algoritmo optimizando bucles y la ordenación de manera que el máximo y el mínimo se calculen mientras se ordenan los puntos, sin embargo, esto no supone una mejora en cuanto al orden de eficiencia y si supone una mayor complejidad del codigo, por lo tanto, como el compilador se encarga de optimizar esos detalles que no afectan al orden de eficiencia, se ha optado por dejarlo de esta forma para facilitar la lectura, en una implementación real, se recomendaría la optimización manual de estos detalles, ya que, aunque bueno, el compilador no es perfecto y no siempre optimiza de la mejor forma.

4.4 Conclusiones

Una vez presentados los tres algoritmos, nos disponemos a obtener conclusiones acerca de ellos así como a compararlos para determinar cual es el mejor, o cómo podríamos mejorar la precisión de los mismos. Comenzamos mostrando visualmente los resultados de ejecución de cada uno de ellos, con el objetivo de que el lector pueda apreciar de forma sencilla la idea que se ha querido transmitir con cada uno de los algoritmos.

4.4.1 Graficas

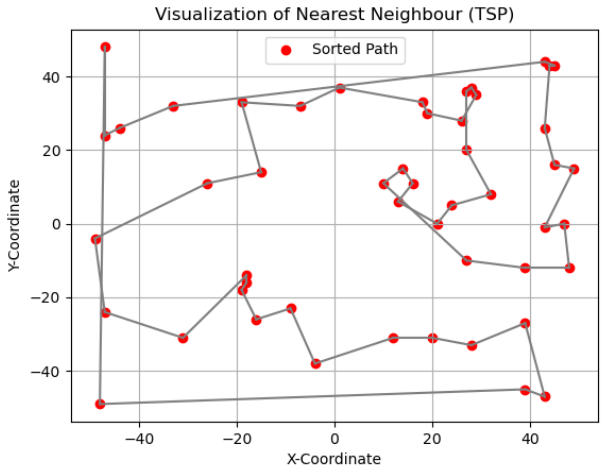

Figura 4.1: Nearest Neighbour

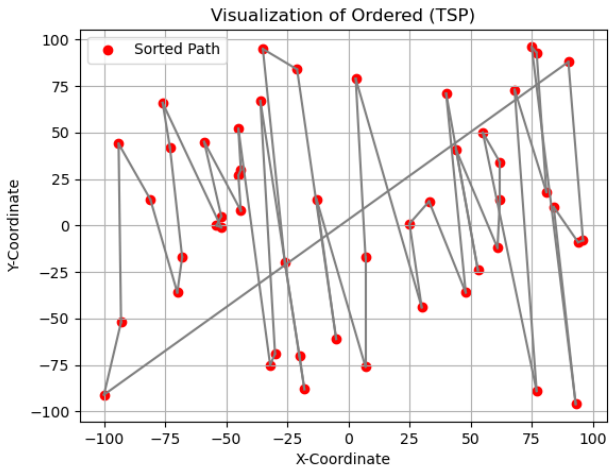

Figura 4.2: Ordenación

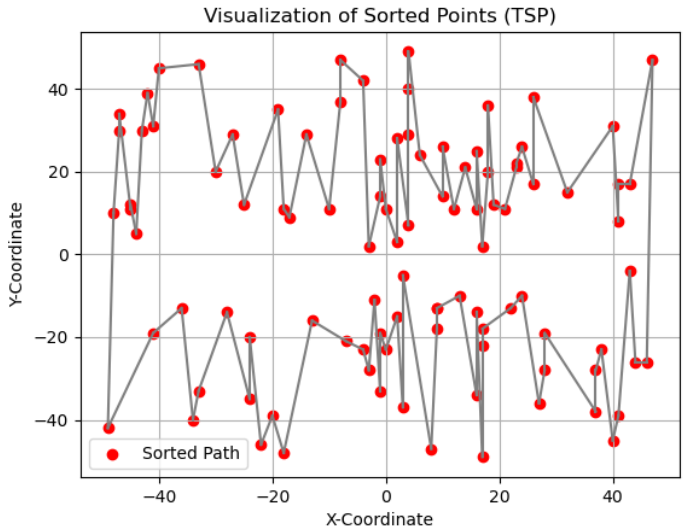

Figura 4.3: Circular

Una vez graficadas, podemos ver los problemas principales de cada uno de los algoritmos, en el caso de *Nearest Neighbour*, se puede apreciar que la distancia de cierre es muy grande, sin embargo, aparenta un camino pare-

cido al óptimo, en el caso de *Ordenación*, se puede apreciar que, mientras los puntos x están cerca, los saltos en y son muy grandes, y además la distancia de cierre es máxima, en el caso de *Circular*, se puede apreciar que los saltos en y son más pequeños y la distancia de cierre es menor.

De este análisis visual, podemos deducir que el algoritmo *Nearest Neighbour* es el que mejor resultado proporciona, le sigue *Circular* y por último *Ordenación*. Podríamos dejar el estudio aquí, sin embargo, se proponen maneras de mejorar los algoritmos.

4.4.2 Mejoras

Para mejorar y comparar los algoritmos, comenzamos notando que en el caso de *Nearest Neighbour*, la elección del punto de partida nos proporciona un camino distinto, por lo que, para mejorar la precisión del algoritmo, se puede ejecutar el algoritmo con distintos puntos de partida y quedarnos con el mejor resultado. En los casos de *Ordenación* y *Circular*, esto no tendría mucho sentido, sin embargo podríamos jugar con la ordenación respecto de un eje u otro, o con la división en dos partes, respectivamente, para mejorar la precisión de los mismos.

Este tipo de mejoras, son sencillas y no suponen un gran aumento en cuanto al tiempo de ejecución, sin embargo, tampoco suponen una mejora significativa en cuanto a la precisión. Su mejor virtud es que nos proporcionan una velocidad de ejecución muy rápida y una precisión aceptable, por lo que, en la práctica, se recomienda su uso, sin embargo, habrá aplicaciones en las que se necesite una precisión mayor, en cuyo caso se recomienda el uso de algoritmos más complejos como λ-opt o genéticos.

4.4.3 Precisión y Tiempos

En esta sección se proporcionan los tiempos de ejecución de los algoritmos y se comparan con la precisión de los mismos, para ello, se han ejecutado los algoritmos sobre un entorno cuya solución óptima se conoce, de esta forma, se puede comparar la solución obtenida con la solución óptima, el entorno en cuestión es un conjunto de Paises del mundo que se han dividido en pueblos y ciudades de interés. A continuación mostramos los resultados que hemos obtenido:

Comenzamos exponiendo los resultados que arrojan los algoritmos sin mejoras:

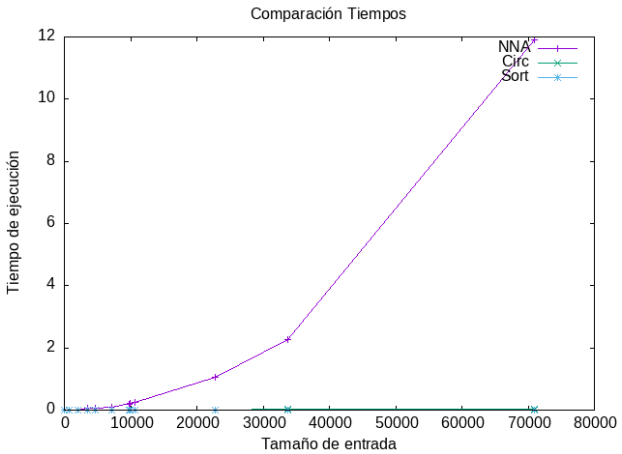

Figura 4.4: Tiempos completos

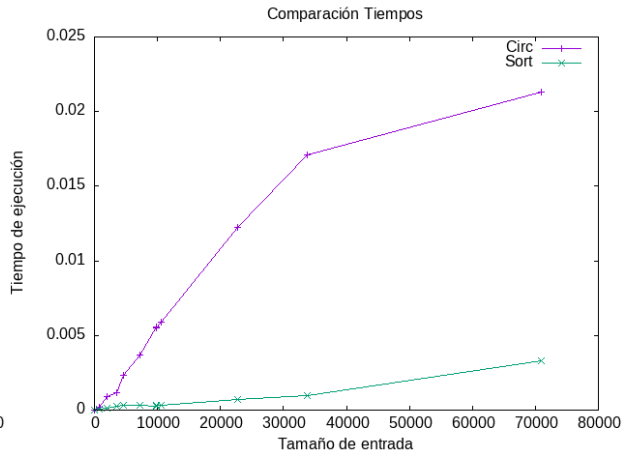

Figura 4.5: Tiempos Circ y Sort

Como se puede ver, el algoritmo *Nearest Neighbour* es el más lento, esto era de esperar debido a que, como

veremos ahora, es el que mejor precisión ofrece. El mismo razonamiento se puede aplicar a los algoritmos *Ordenación* y *Circular*. Si nos fijamos ahora en los tiempos de ejecución de los algoritmos tras las mejoras,

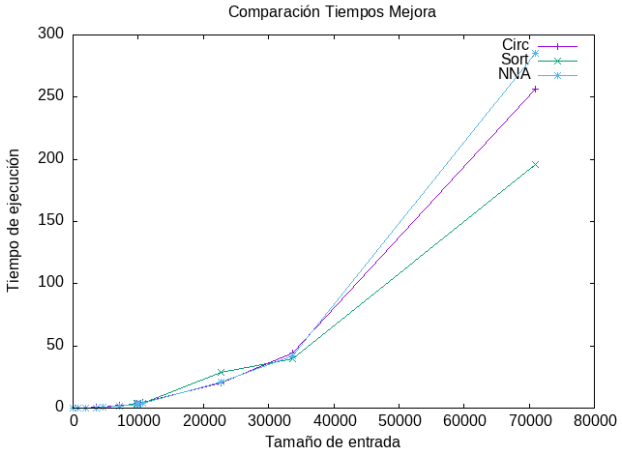

Figura 4.6: Tiempos completos

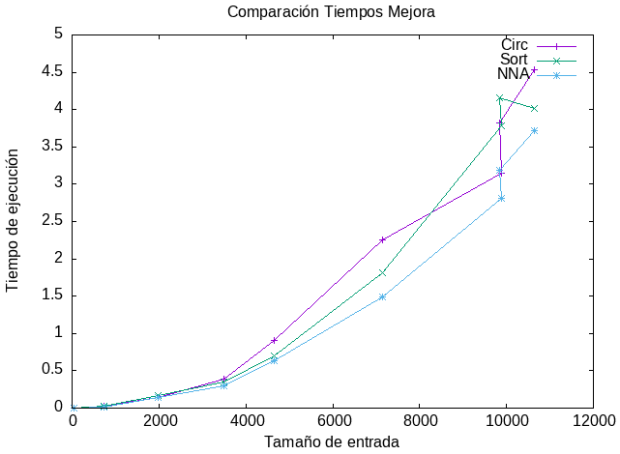

Figura 4.7: Tiempos menor Tamaño

Podemos ver como hay un gran aumento en el tiempo, sin embargo, son tiempos similares entre ellos, por lo que nos interesa quedarnos con el que mejor precisión ofrece:

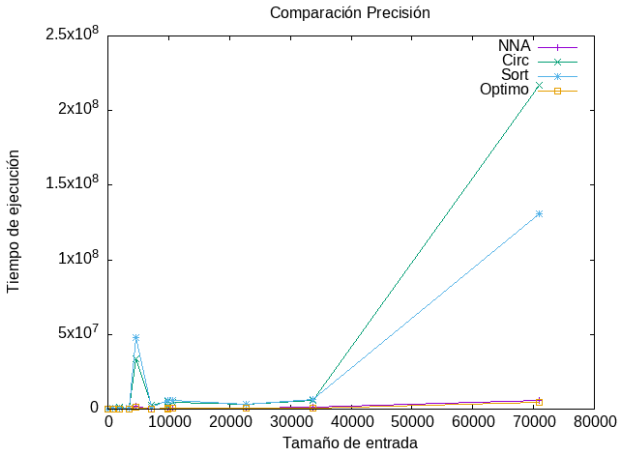

Figura 4.8: Precisión sin mejoras

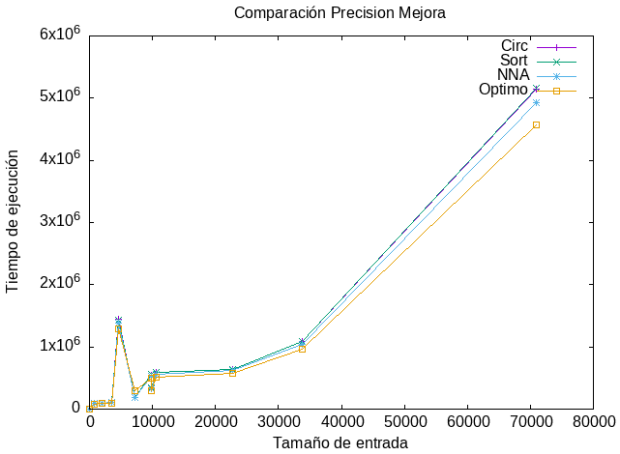

Figura 4.9: Precisión tras mejora

Como se puede ver, todos los algoritmos con mejora, se quedan muy cerca de la solución óptima, sin embargo, el algoritmo *Nearest Neighbour* es el que mejor precisión ofrece, tanto en el caso de mejora como en el caso sin mejora, por lo que, en la práctica, se recomienda su uso. Para aplicaciones donde la precisión sea un factor crítico, se recomienda el uso combinado de distintas mejoras como la elección de distintos puntos de partida o el uso de algoritmos más complejos como λ-opt o genéticos.