Problema de secuenciación de tareas en máquinas paralelas con objetivo de latencia.

Francisco Javier Mendoza Álvarez - alu0100846768@ull.edu.es
David de León Rodríguez - alu0100965667@ull.edu.es
Carlos García González - alu0100898026@ull.edu.es
Aduanich Rodríguez Rodríguez - alu0100818130@ull.edu.es
Alejandro David Carrillo Padrón - alu0100845808@ull.edu.es

Índice

Introducción

El problema de secuenciación de tareas con objetivo de latencia es un contratiempo común en múltiples ámbitos, no solo en el campo de la informática, es por esto que es interesante abordar este problema.

Proceso para la toma de decisiones que se centra en la asignación de recursos a tareas en periodos temporales determinados.

Secuenciación

Descripción del problema

Objetivo

Tratar de encontrar una distribución de las tareas a ejecutar entre las máquinas disponibles de forma que la suma de latencias de las tareas sea mínimo.

Problema

MAX

Número de Tareas por máquina.

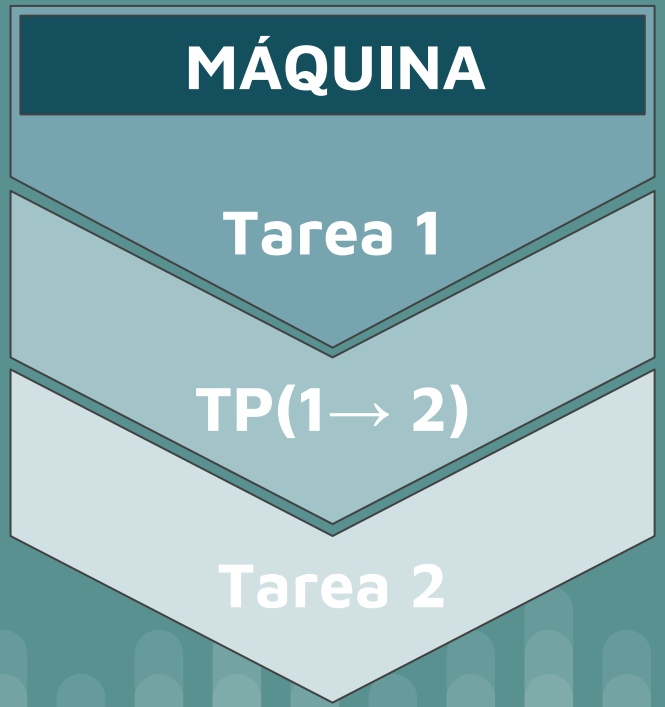
MIN

Latencia Total.

Descripción del problema

M Máquinas
N Tareas

Tenemos que introducir las N tareas en las M máquinas de forma que se minimice la suma de la latencia de las máquinas.



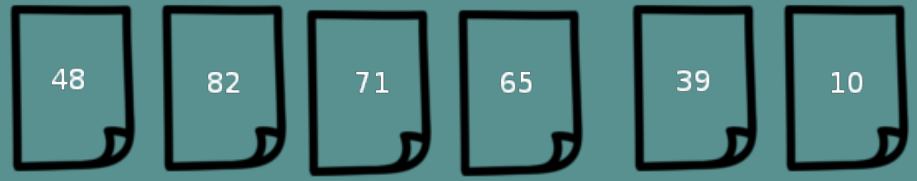
Descripción del problema

Latencia

Tiempo que espera cada tarea a ser ejecutada más el tiempo que tarda en ejecutarse.

Latencia Total

Suma de los tiempos de latencia de todas las tareas asignadas a esa máquina en el orden de ejecución correspondiente más los tiempos de espera concretos entre las ejecuciones de cada una de las tareas.



50 + (50 + (36 + 35)) + (50 + 36 + 35 + (40 + 78)) = 410

59 + (59 + (20 + 88)) + (59 + 20 + 88 + (18 + 21)) = 432

Latencia total
842

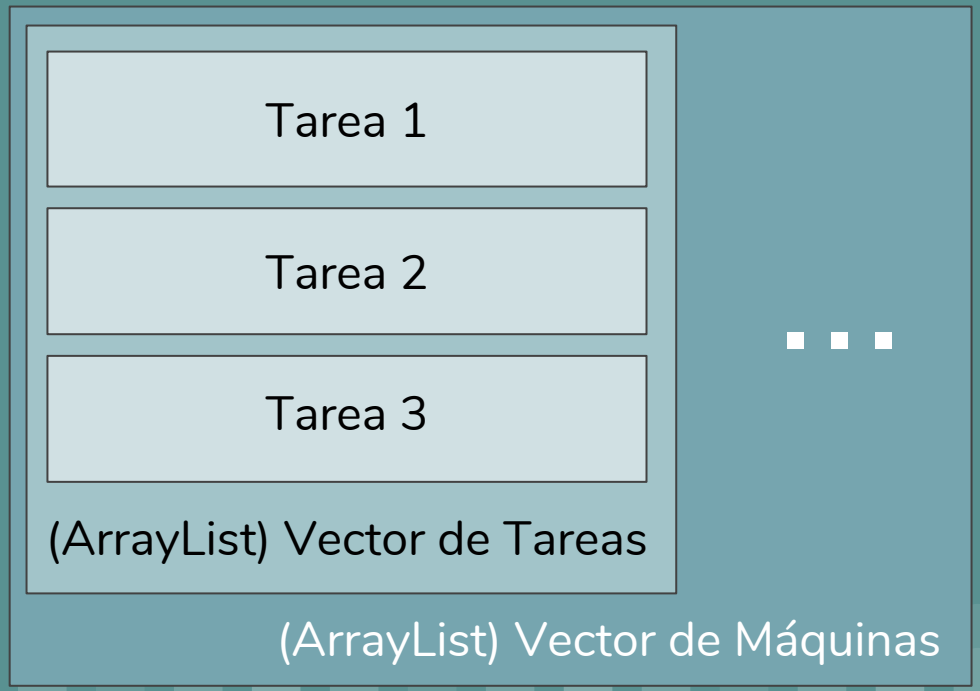
Función Objetivo

$$LatenciaTotal = \sum_{i=1}^{m}(P_0 + \sum_{j=1}^{n}(S_{jk} + P_k))$$

Donde:

m = Número de máquinas.
n = Número de tareas de la máquina “m”.
Px = Tiempo de ejecución de la tarea “x”.
Sjk = Tiempo de preparación entre la tarea “j” y la tarea “k”.

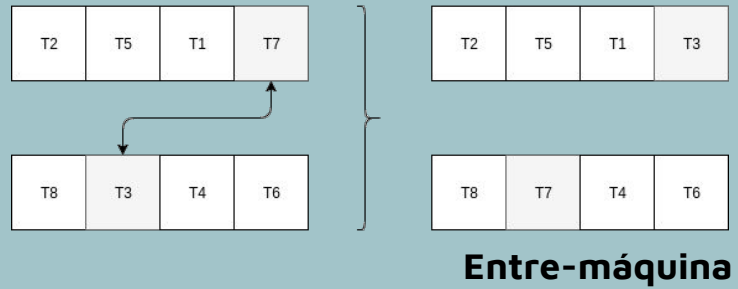
Estructura de la solución



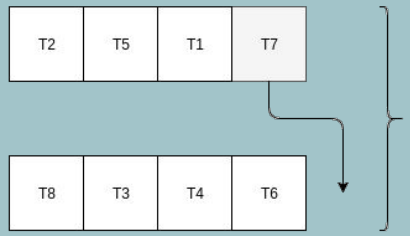
Tiempos de preparación
Integer [i][j]
Tiempo de preparar tarea “j” después de la “i”.

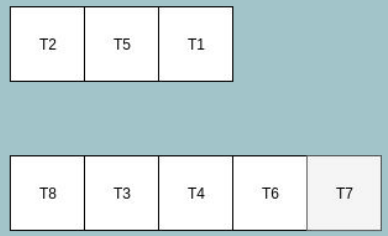
Estructuras de entorno

- **Primero Mejor**
- **Greedy**

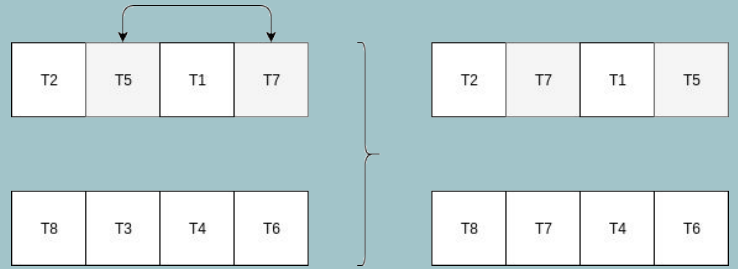


Entre-máquina





Reinserción



Intra-máquina

Algoritmos - Greedy

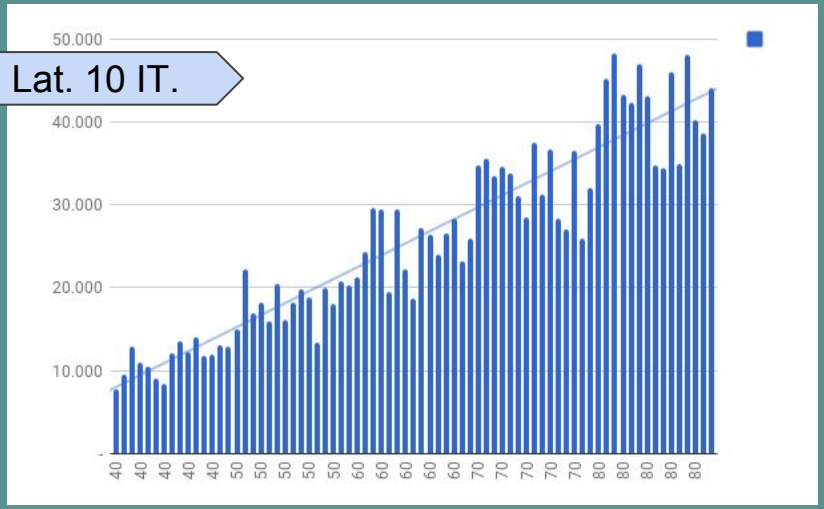
EXPLICACIÓN

Mientras queden tareas que insertar, comprueba la máquina con mejor latencia y la inserta dentro de la misma en la mejor posición intra-máquina posible.

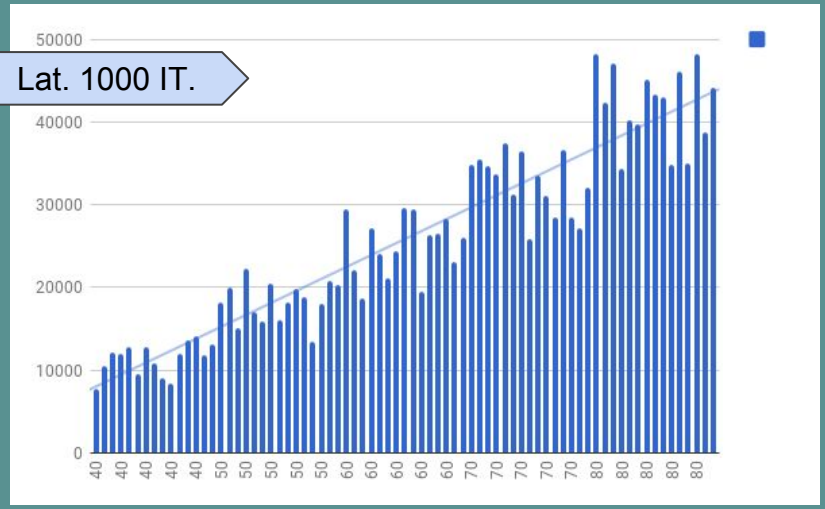
CODE

```
exec(){
     while (TasksToProcess > 0){
          x = getBestMachine();
          for(i=0; i<x.getTareas().size(); i++){
               x.add(i, TasksToProcess.get(0));
               if(actualLatency > x.getLatency()){
                    actualLatency = x.getLatency();
                    bestIndex = i;
               }
               else{
                    x.remove(i);
               }
          }
          x.add(bestIndex, TasksToProcess.get(0));
     }
}
```

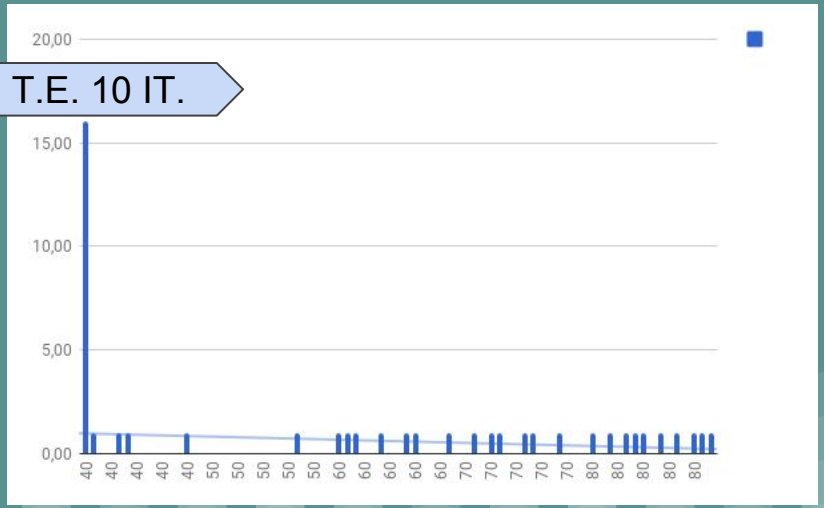
Lat. 10 IT.



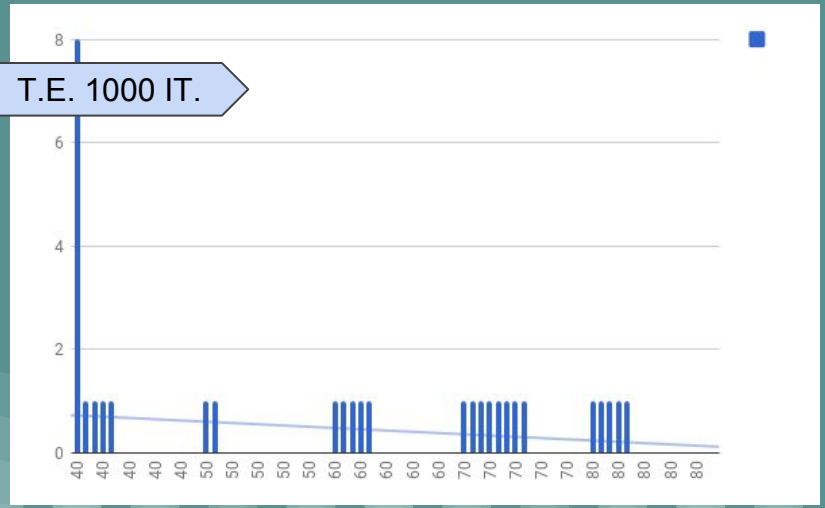
Lat. 1000 IT.



T.E. 10 IT.



T.E. 1000 IT.



Algoritmos - Grasp

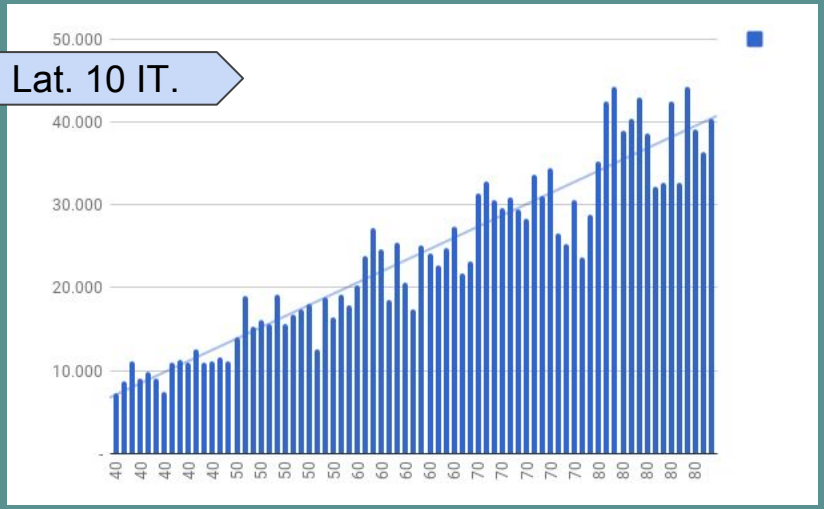
EXPLICACIÓN

Mientras queden tareas que insertar, genera una lista restringida de candidatos entre los X primeros elementos de “TasksToProcess” e inserta uno de ellos al azar de la mejor forma usando para ello un algoritmo Greedy.

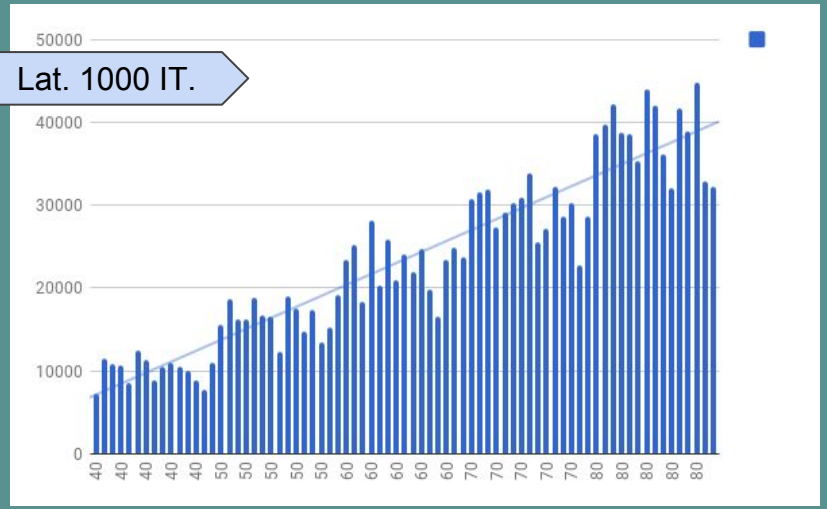
CODE

```
exec(){
     buildSolution();
     upgrade();
}
buildSolution(){
     while (TasksToProcess > 0){
          Random rnd(0 → LRC.size());
//Greedy adding random task 'TasksToProcess.get(rnd)'
          x = getBestMachine();
          y = getBestPosOnMachine(x);
          x.add(y, TasksToProcess.get(rnd));
     }
}
upgrade(){
//Search in local neighborhood for a better latency
}
```

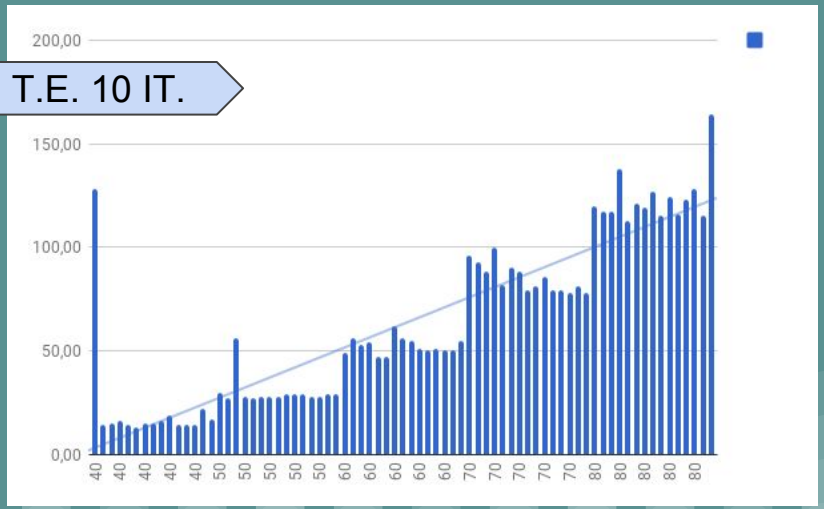
Lat. 10 IT.



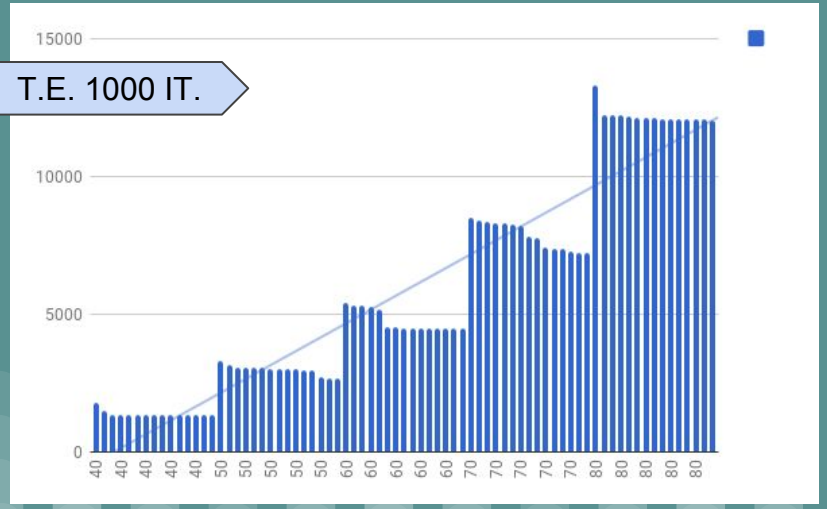
Lat. 1000 IT.



T.E. 10 IT.



T.E. 1000 IT.



Algoritmos - Multiarranque

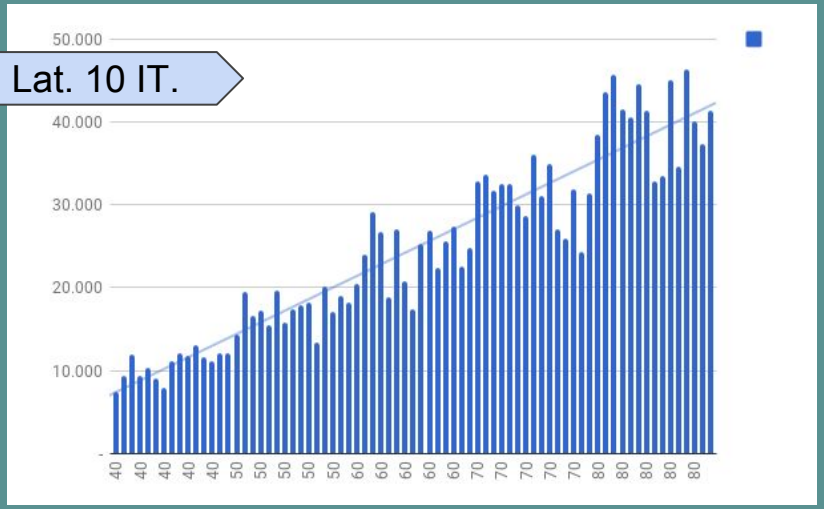
EXPLICACIÓN

Generamos las “N” soluciones más dispares posibles usando un algoritmo aleatorio y mejoramos dichas soluciones usando búsquedas locales devolviendo la solución con menor latencia de estas.

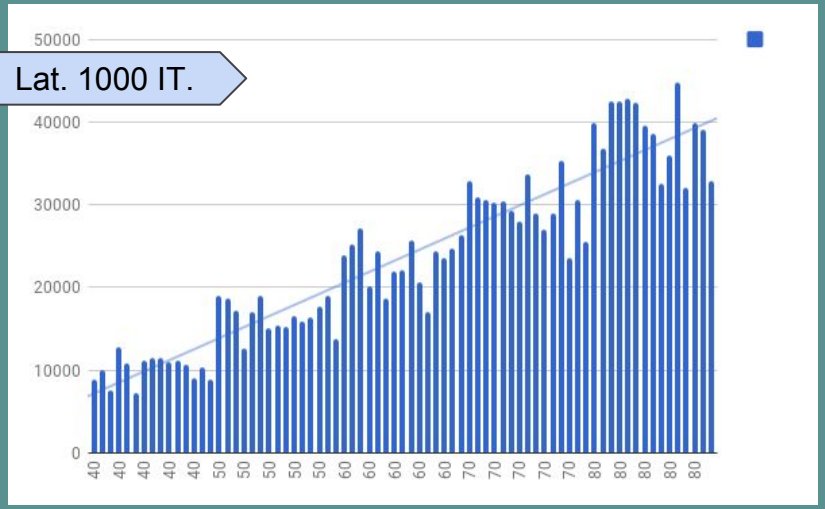
CODE

```
//N_WANTED_SOLUTIONS = number of different
solutions to start searching
exec(){
     ArrSols = new Solutions[N_WANTED_SOLUTIONS];
     for(i = 0; i<N_WANTED_SOLUTIONS; i++){
          //Using any random Algorithm
          x = buildSolution();
          //Using any local search
          x.upgrade();
          ArrSols.add(x);
     }
     return getBestSolution(ArrSols);
}
getBestSolution(Solutions[]){
     //return Solution with better latency (for)
}
```

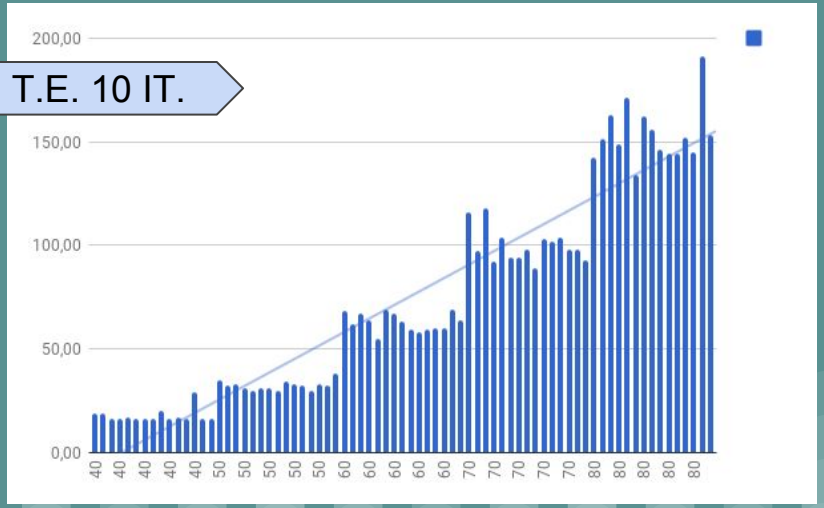
Lat. 10 IT.



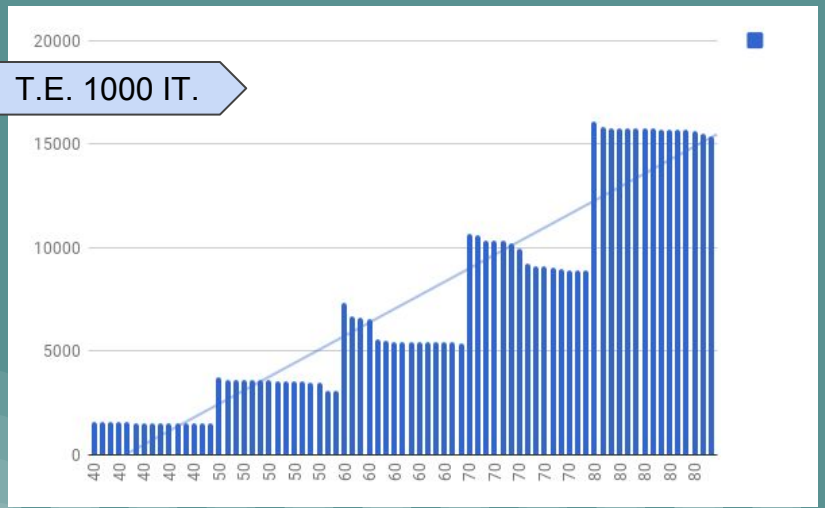
Lat. 1000 IT.



T.E. 10 IT.



T.E. 1000 IT.



Algoritmos - VNS

EXPLICACIÓN

El VND recorre las diferentes estructuras de entorno, generando un vecino. En caso de que sea mejor se vuelve a la estructura de entorno inicial y se actualiza la solución. En caso contrario se pasa a la siguiente estructura de entorno.

CODE

```
vnd(){
     entorno = 0;
     while(entorno < N_ENTORNOS){
          y = x.getNeighbour(entorno);
          if(y.getLatency() <
x.getLatency){
               x = y;
               entorno = 0;
          }
          else entorno++;
     }
}
```

Algoritmos - VNS

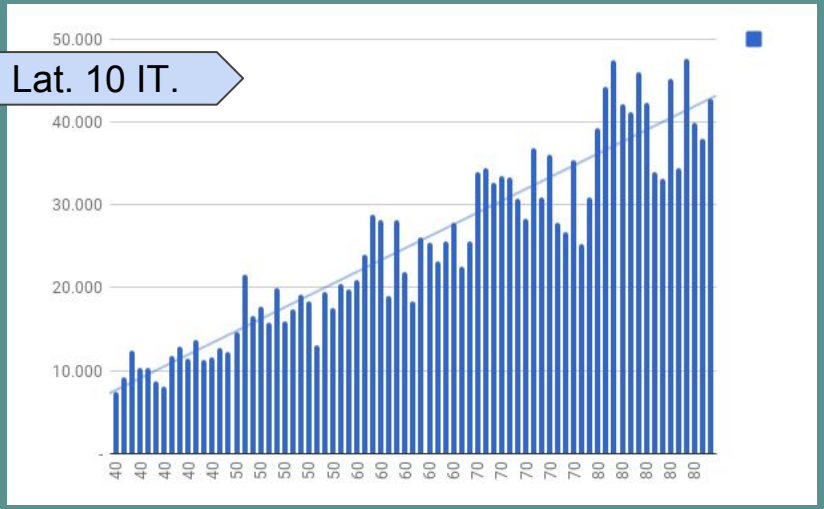
EXPLICACIÓN

El VNS recorre también las estructuras de entorno, alterando la solución actual dentro del entorno actual de forma aleatoria (shake). Tras esto mejoramos esta solución con una búsqueda local o un vns y comprobamos si es mejor que la solución conseguida anteriormente.

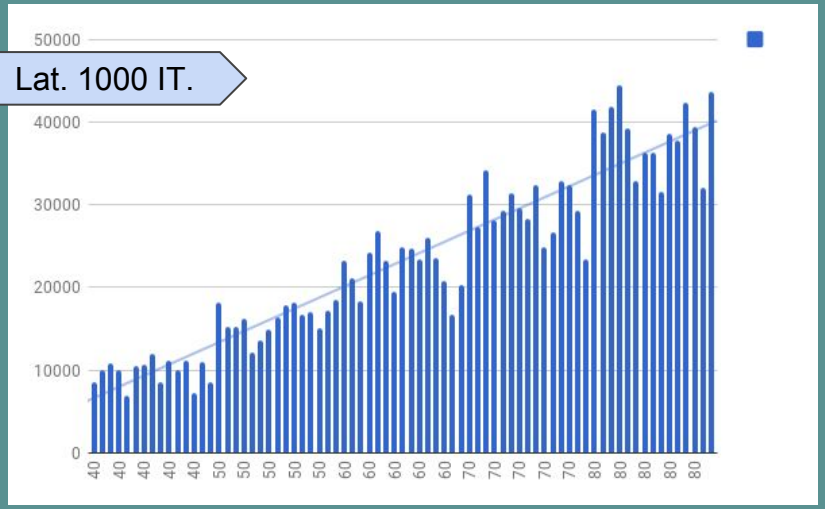
CODE

```
exec(){
     x = buildSolution(); //Using random Algorithm
     for(i = 0; i < N_ITERACIONES; i++){
          entorno = 0;
          while(entorno < N_ENTORNOS){
               y = x
               y.shake();
               y.vnd(); //Or local search
               if(y.getLatency()< x.getLatency()){
                    x = y;
                    entorno = 0;
               }
               else entorno++;
          }
     }
}
```

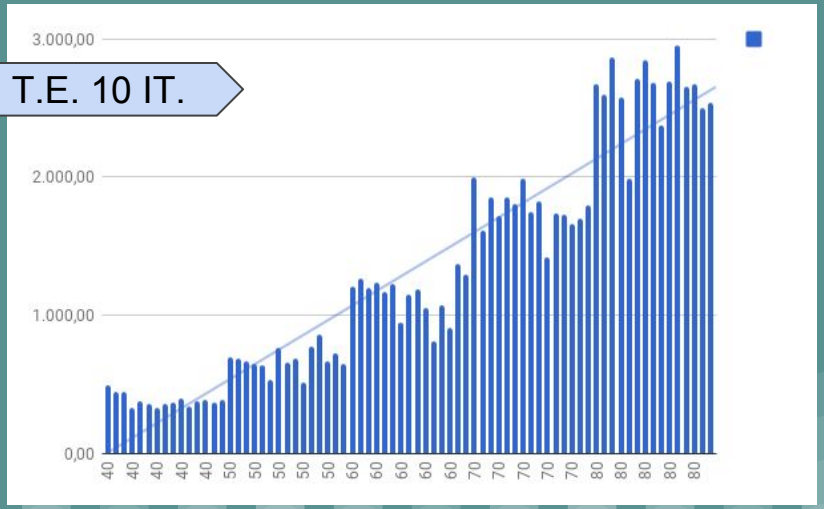
Lat. 10 IT.



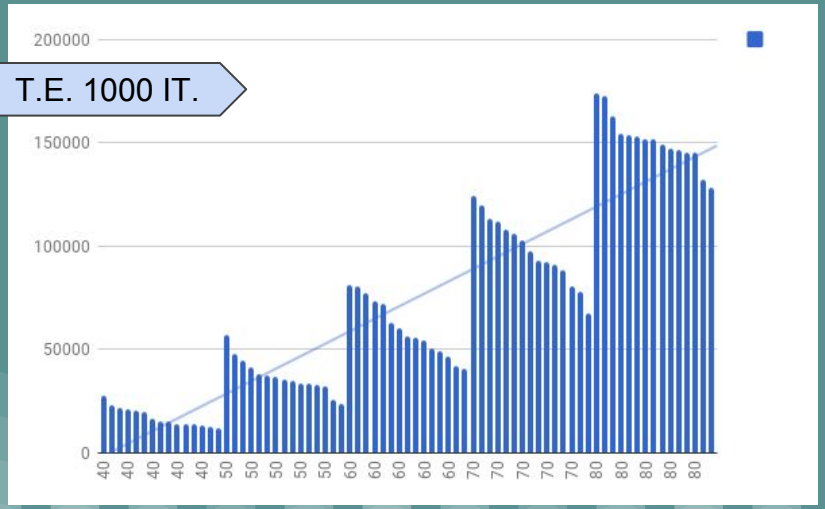
Lat. 1000 IT.



T.E. 10 IT.



T.E. 1000 IT.



Algoritmos - Tabú Search

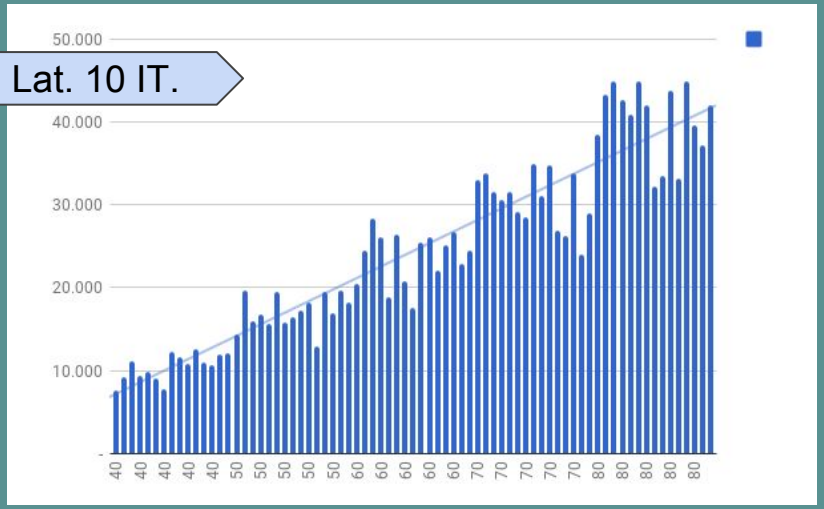
EXPLICACIÓN

Se construye una solución, se busca el mejor cambio posible entre los vecinos, si este no está en la lista tabú, se añade eliminando el cambio más viejo en caso de que la lista esté llena y se ejecuta el cambio. Posteriormente se comprueba si mejora la solución o no.

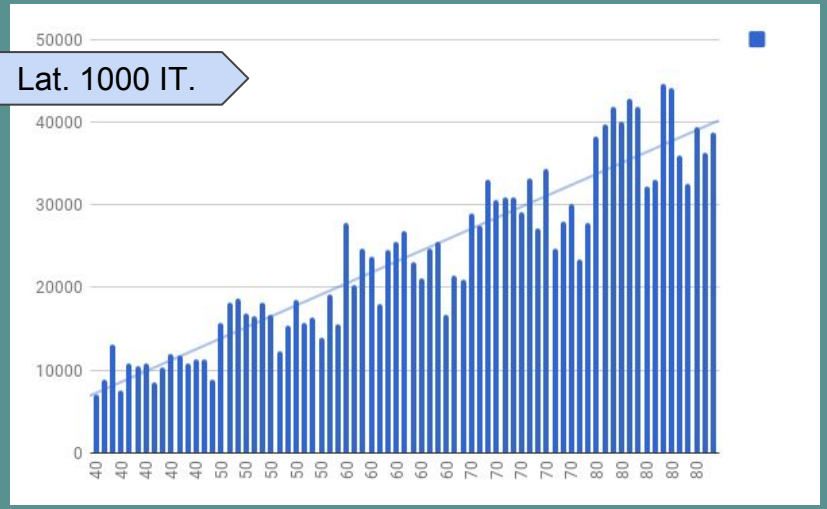
CODE

```
exec(){
      x = buildSolution();//Using random Algorithm
      //Using between all neighborhood struct except
      reinsertion
      for(i = 0; i < N_ITERATIONS; i++){
            y = x.getBestNeighborhoodChange(tabuList);
            if(!tabuList.contains(y)){
                  if(tabuList.size() > MAXTABULISTSIZE)
                        tabuList.remove(0);
                  tabuList.add(y);
                  x.upgrade(y);
                  if(actualLatency > x.getLatency()){
                        actualLatency = x.getLatency();
                        bestSolution = x;
                  }
            }
      }
}
```

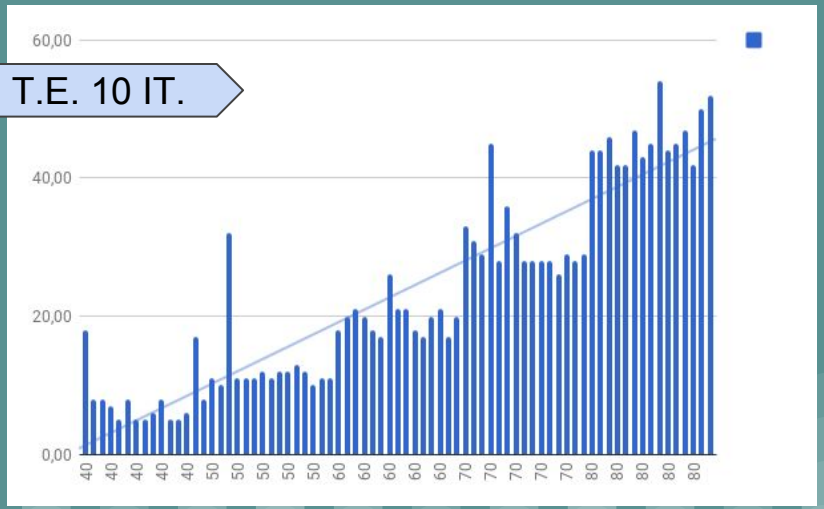
Lat. 10 IT.



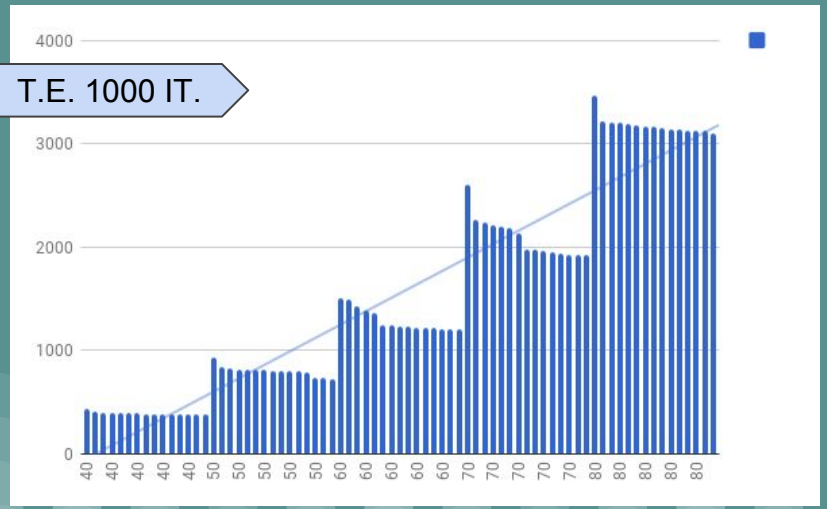
Lat. 1000 IT.



T.E. 10 IT.



T.E. 1000 IT.



Algoritmos - LNS

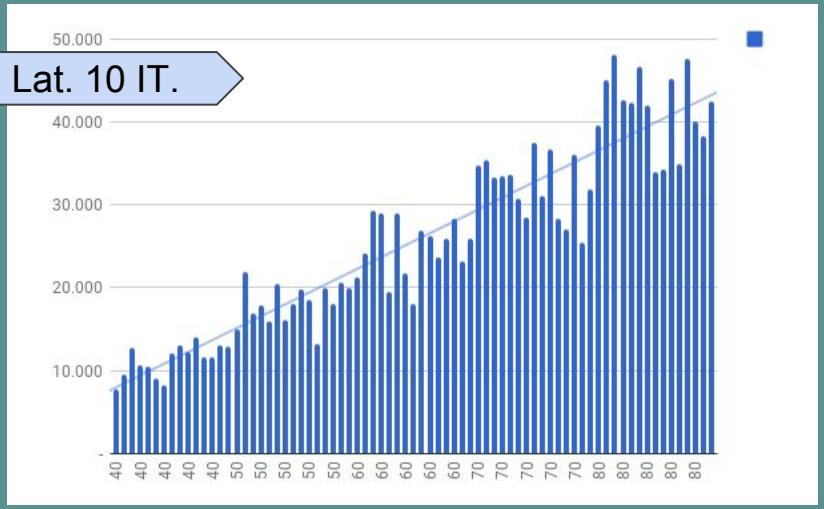
EXPLICACIÓN

Se genera una solución aleatoria inicial y se destruye un porcentaje de la misma aleatoriamente para reconstruirla usando una estructura de entorno y comprobar si se ha alcanzado una solución mejor. En dicho caso se reinicia el porcentaje de destrucción, mientras que en caso opuesto, este aumenta.

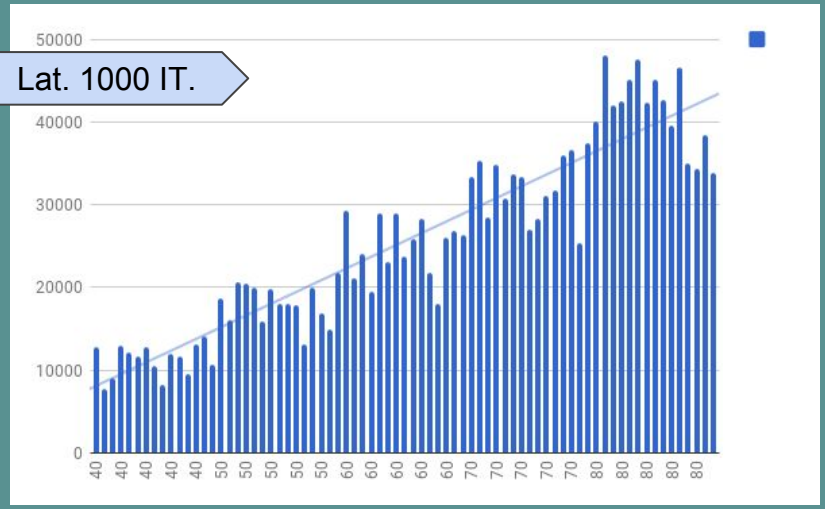
CODE

```
exec(){
      x = buildSolution(); //Using random Algorithm
      while(!stopCondition){
      //Destroying a rate of random elements
            x.destroySolution(destroyRate);
      //Using between all neighborhood struct except
      reinsertion
            x.reconstructSolution();
            x.upgrade();
            if(actualLatency > x.getLatency()){
                  actualLatency = x.getLatency();
                  bestSolution = x;
                  restartDestroyRate();
            }
            else{ increaseDestroyRate(); }
      }
}
```

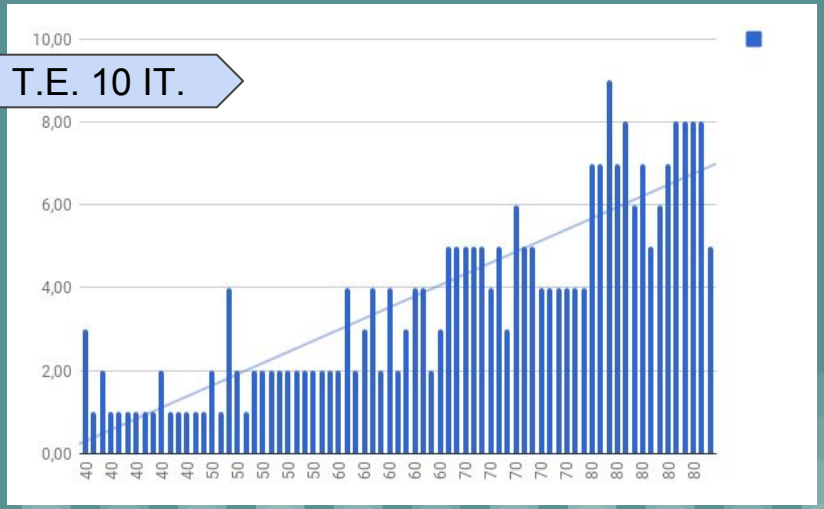
Lat. 10 IT.



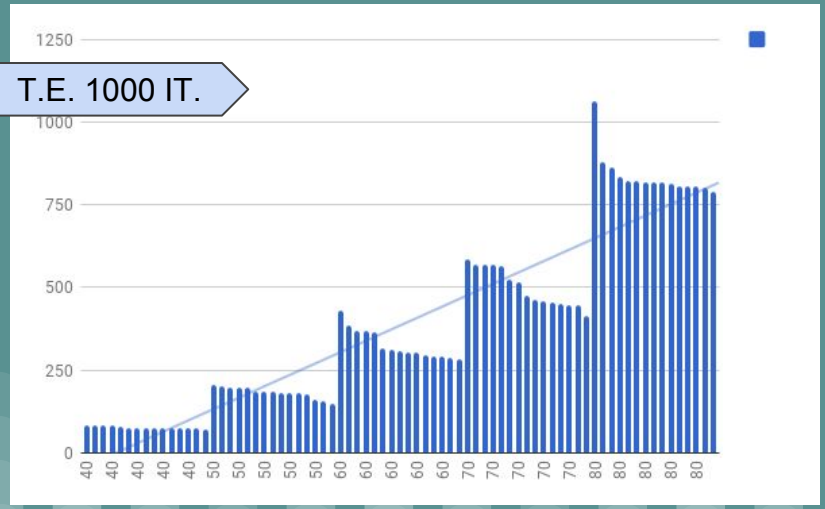
Lat. 1000 IT.



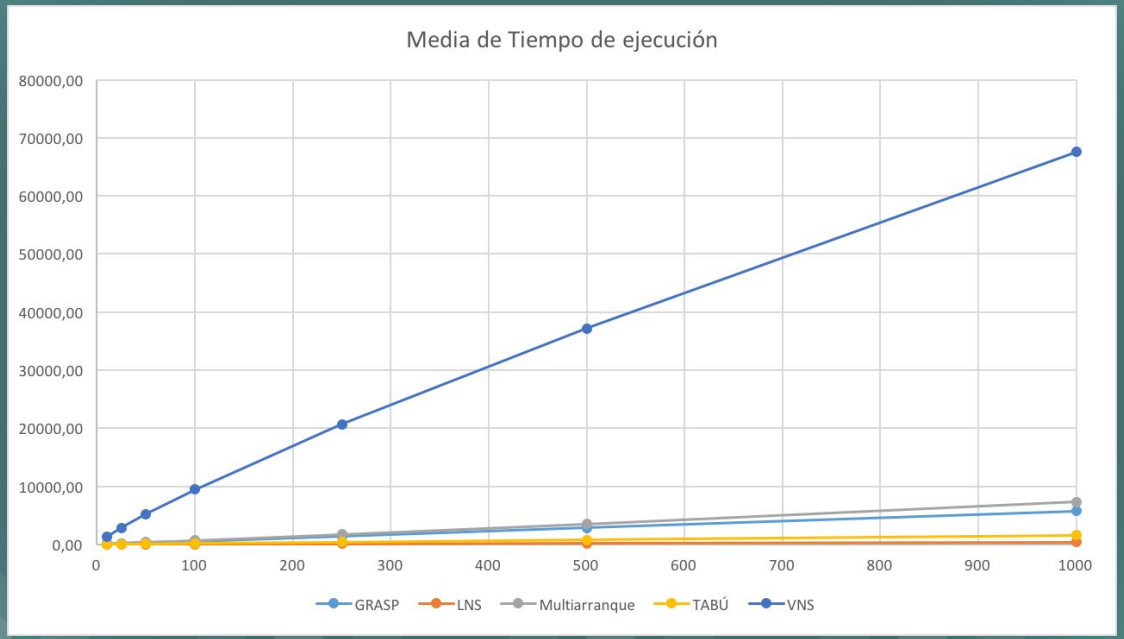
T.E. 10 IT.



T.E. 1000 IT.

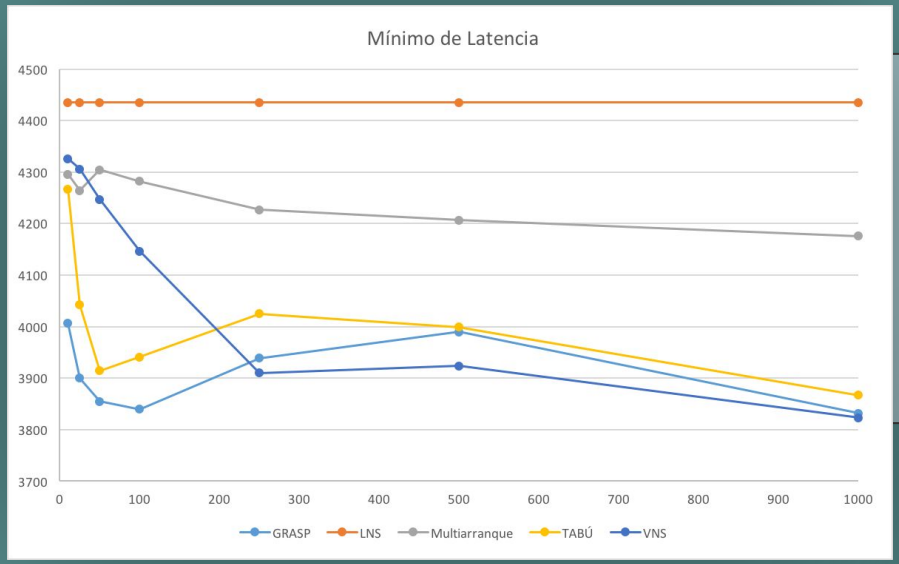


Comparativa de Algoritmos



T. Medio Ejecución

- LNS
- TABÚ
- GRASP
- MULTIARRANQUE
- VNS

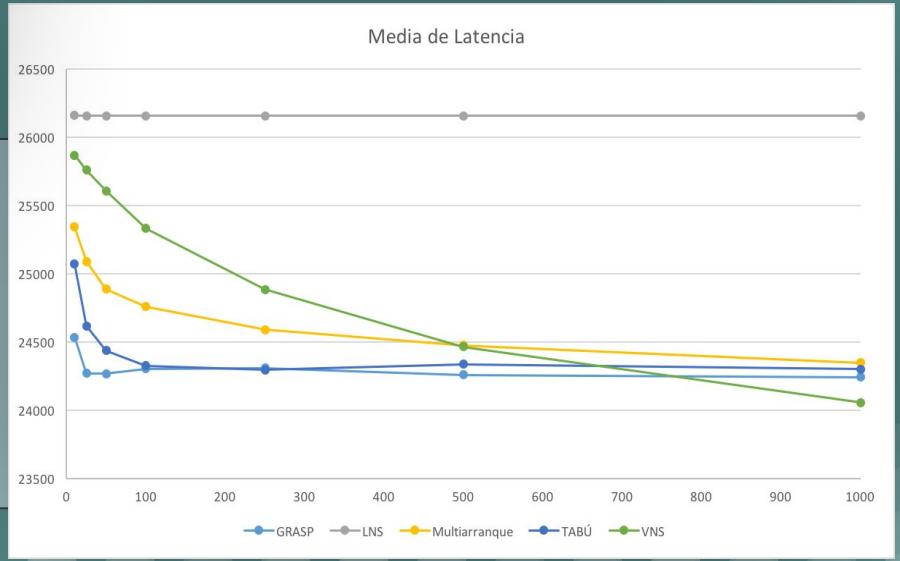


Latencia Mínima (Mejor resultado)

- VNS
- GRASP
- TABÚ
- MULTIARRANQUE
- LNS

Latencia Media

- VNS
- GRASP
- TABÚ
- MULTIARRANQUE
- LNS



Conclusiones

1

EL ALGORITMO CON MEJOR RELACIÓN LATENCIA/TIEMPO (RESULTADO) ES EL TABÚ SEGUIDO DEL GRASP

2

CON EL TIEMPO SUFICIENTE EL ALGORITMO QUE MEJORES RESULTADOS DEVUELVE ES EL VNS

3

DEBIDO AL FACTOR ALEATORIO DE DESTRUCCIÓN DEL LNS, ES EL ALGORITMO QUE PEOR RESULTADOS HA DEVUELTO

¿Preguntas?

Gracias por su atención