UNIVERSIDAD DE GRANADA

Algorítmica

Algoritmos Voraces:
Sala de reuniones

UNIVERSIDAD DE GRANADA

ETSIIT
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Juan Miguel Acosta Ortega

UNIVERSIDAD
DE GRANADA

Índice

UNIVERSIDAD DE GRANADA

1. Problema inicial : Sala de reuniones

Una empresa dispone de una única sala de reuniones. Se tienen previstas “n” reuniones, cada una con unos tiempos de inicio y de finalización preestablecidos, (ini[i],fin[i]). Se desea conocer el número máximo de reuniones que se pueden realizar en la sala (y cuales son), suponiendo que no se pueden realizar varias reuniones simultáneamente. Diseñando un algoritmo voraz para resolver este problema y demostrad su optimalidad:

1.1 Diseño Greedy del problema

Un algoritmo greedy (o voraz) es un tipo de algoritmo de optimización que busca tomar decisiones óptimas en cada etapa de un problema, basándose en la elección que maximice o minimice el resultado inmediato sin tener en cuenta las consecuencias a largo plazo. Es decir, en cada paso, el algoritmo toma la mejor decisión posible en función de la información disponible en ese momento, sin considerar cómo esa decisión afectará a futuros pasos del proceso.

A menudo, los algoritmos greedy se utilizan para resolver problemas de optimización en los que se busca encontrar la solución más óptima posible, aunque no garantizan encontrar la solución global óptima en todos los casos. La estrategia greedy puede ser útil para solucionar problemas de manera rápida y eficiente en casos en los que la solución global no es necesaria o es demasiado costosa en términos de tiempo y recursos.

El diseño de este problema concreto por parte del grupo teniendo en cuenta que el objetivo es maximizar el número de reuniones es el siguiente:

Lista de candidatos: Reuniones asignadas a una sala (Supongamos que a lo largo de un día).

Lista de candidatos utilizados: Reuniones ya terminadas.

Función solución: En este caso no hay una solución clara como en el problema de la máquina expendedora que devuelve cambio; mientras que ese caso tiene la función solución (punto de parada) de llegar a un cambio exacto, **este problema parará de buscar cuando la lista de candidatos esté vacía o cuando ya no haya reuniones factibles (según función factibilidad).**

Criterio factibilidad: Una reunión podrá ser insertada en la solución siempre y cuando su inicio sea mayor que el inicio de la anterior y que el final de la anterior también, y que su inicio y final sean menores que el inicio de la siguiente: **((INI [x] > INI [y] && INI [x] > FIN [y]) || (INI [x] < INI[y] && FIN[x] < INI[y])).**

Función selección: La selección inmediata de una reunión de la lista de candidatos se puede realizar comprobando varias cosas : **la que comience antes , la más corta (poco óptimas ambas), la que menos se solape o la que antes termine.**

Spoiler: La que garantiza la solución óptima siempre es la que selecciona primero la que termine antes.

Se implementarán estas funciones selecciones y se discutirá cuál de ellas es la más óptima.

Función objetivo: Maximizar el número de reuniones en un día.

Se diseña el algoritmo con pseudocódigo según lo razonado:

```
funcion R = Greedy ( vector candidatos C){

    R [];

    qs_segun_tipo_seleccion(C,low,high);

    while (!C.empty()){
        x = C.seleccion();
        if (x.factible()){
            R += x;
        }
        C.remove(x);
    }

    return R;
}
```

Se procede siguiendo el esquema de un algoritmo voraz:

- Se parte de un conjunto de "solución" vacío.
- Se ordenan los candidatos según el tipo de función selección (Empieza antes, más corta antes …), por ejemplo si el tipo de selección es el de seleccionar primero a la reunión que termina antes, se ordena el conjunto de manera creciente según ese dato.
- Se selecciona el mejor candidato posible del vector de candidatos (el primero), y se introduce a la solución “R” si cumple el criterio de factibilidad.
- Si no se puede continuar devolvemos la solución obtenida.

Hemos implementado la solución similar a este problema en c++ con el objetivo de probar ciertos casos (no es 100% fiel ya que necesitábamos un programa práctico que solucionara el problema, no hemos hecho la solución más eficiente) y comprobar la optimalidad de las soluciones ofrecidas:

Lista de candidatos / solución: Para esto se ha implementado un vector de struct de “reuniones” con su hora de inicio y fin.

```
 7   struct reunion
 8   {
 9       float inicio;
10       float fin;
11   };
12
13   bool operator==(const reunion& lhs, const reunion& rhs) {
14       return lhs.inicio == rhs.inicio && lhs.fin == rhs.fin;
15   }
```

Además se ha sobrecargado el operador de igualdad para comprobar la igualdad entre reuniones. (Ayuda a poder eliminar de una manera rápida reuniones del vector cuando estén en el vector de soluciones).

Función solución: Se interpreta que se ha llegado a una solución cuando o no quedan candidatos entre los que elegir, o todos los candidatos se solapan con las soluciones actuales.

```
94      while ((c.empty() == false)
```

Criterio de factibilidad: Si el candidato seleccionado no solapa con ninguna reunión, se inserta en el conjunto solución.

```
46
47  bool esFactible(const reunion &r1, const reunion &r2)
48  {
49      return ((r1.inicio > r2.inicio && r1.inicio > r2.fin) || (r1.inicio<r2.inicio && r1.fin < r2.inicio));
50  }
51
52  bool esFactible(const reunion &r1, const vector<reunion> &r)
53  {
54      bool factible = true;
55
56      if (r.empty())
57      {
58          return true;
59      }
60
61      for (int i = 0; i < r.size() && factible == true; i++)
62      {
63          if (!esFactible(r1, r[i]))
64          {
65              factible = false;
66          }
67      }
68
69      return factible;
70  }
```

Función selección: Junto con la función factibilidad, esta es la que decide cómo de óptima puede ser la solución dependiendo del caso inicial de reuniones.

Se han implementado 4 diferentes;

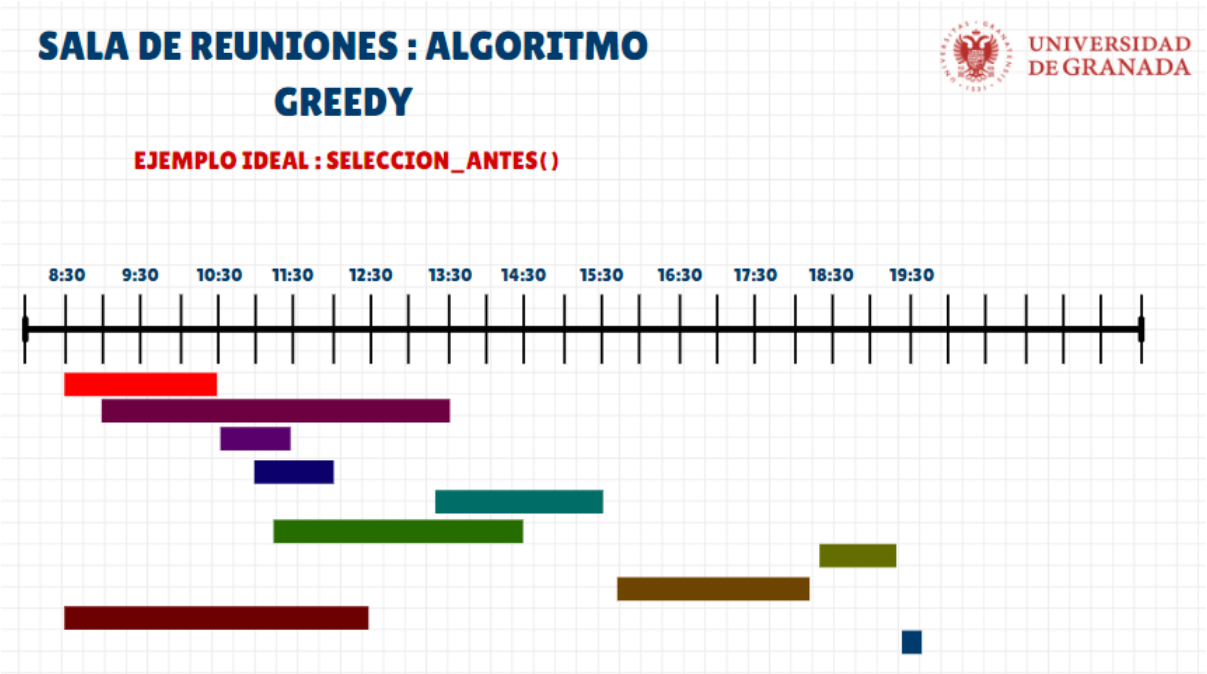
- **seleccionAntes():** Selecciona primero la que comienza antes.
- **seleccionCorta()**: Selecciona primero la reunión más corta.
- **seleccionAntesFin():** Selecciona primero la reunión que termine antes.
- **SeleccionMenosSolapa():** Selecciona primero la reunión que menos se solapa con otras.

```
32   reunion seleccionAntes(const vector<reunion> &c)
17   reunion seleccionCorta(const vector<reunion> &c)
47   reunion seleccionAntesFin(const vector<reunion> &c)
62   reunion seleccionMenosSolapa(const vector<reunion> &c){
```

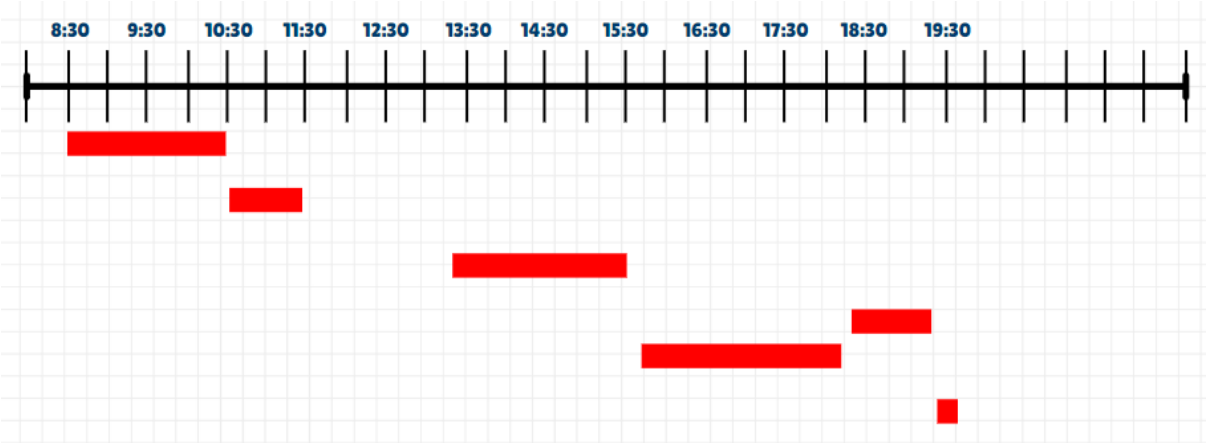
1.2 ¿Es óptima la solución que ofrece el algoritmo voraz?

Teniendo la implementación sobre la mesa, podemos comprobar las soluciones arrojadas por el algoritmo con un ejemplo práctico.

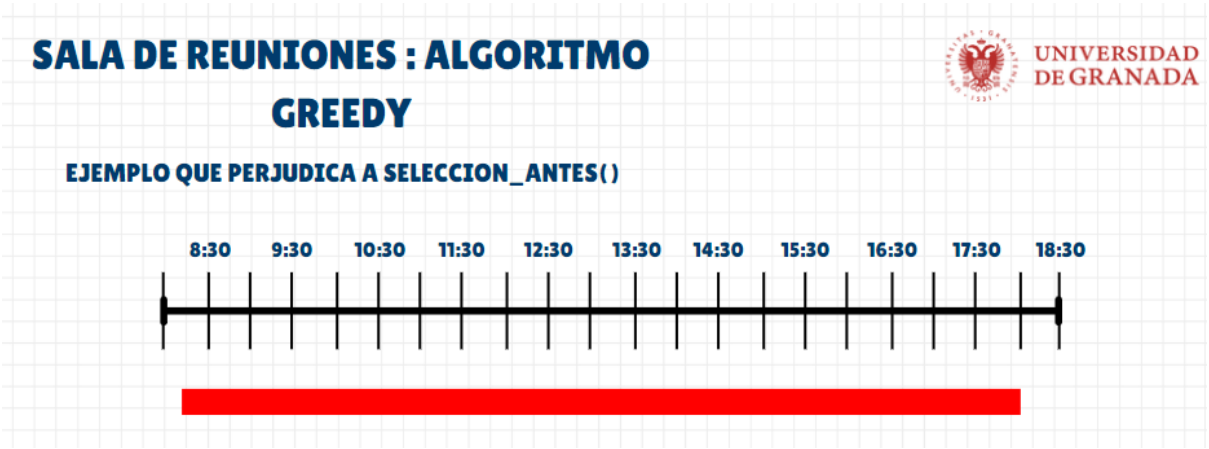
En primer lugar planteamos un ejemplo bueno sobre todo para la función selección en la que empezamos seleccionando las reuniones más tempranas:



De manera rápida y visual podemos ver que la solución es la siguiente, y que ofrece 6 reuniones sobre 10:

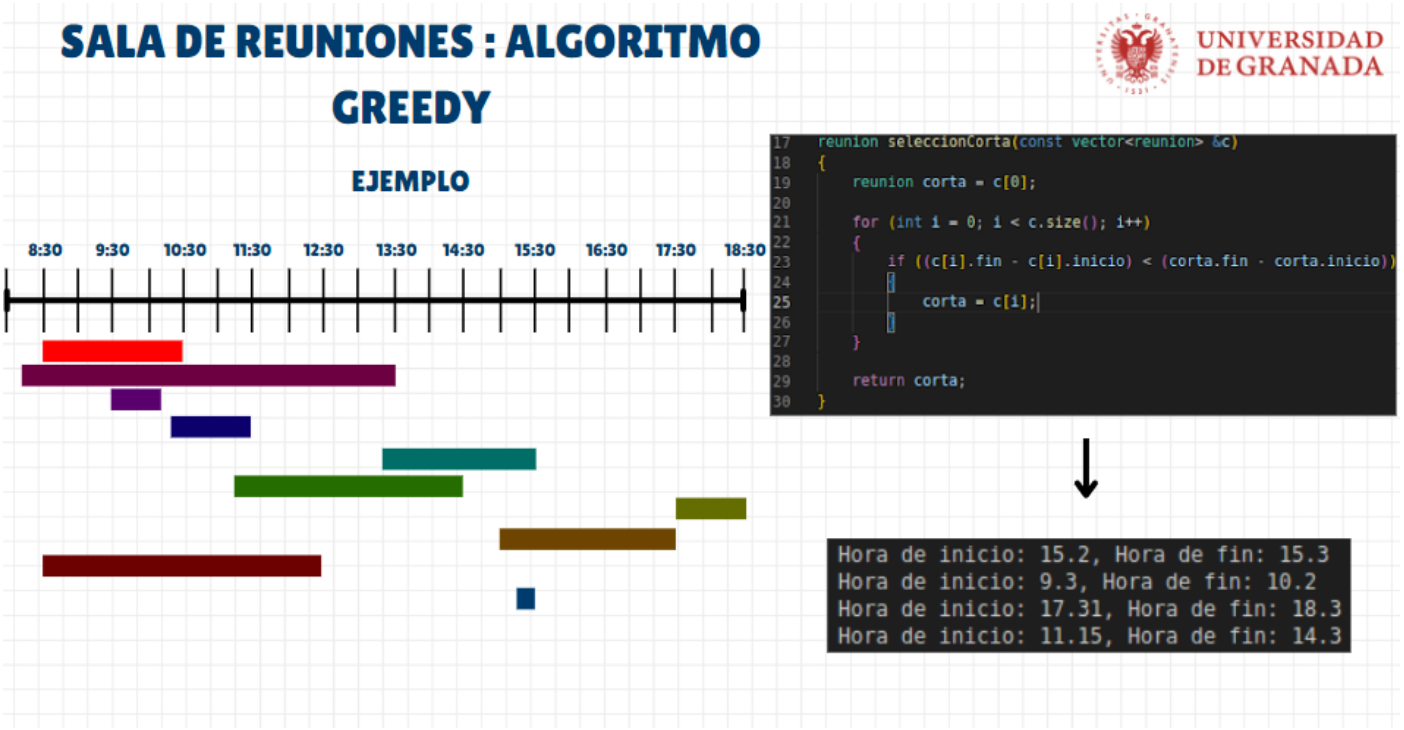


Sin embargo, probemos el algoritmo en un entorno más “hostil”:



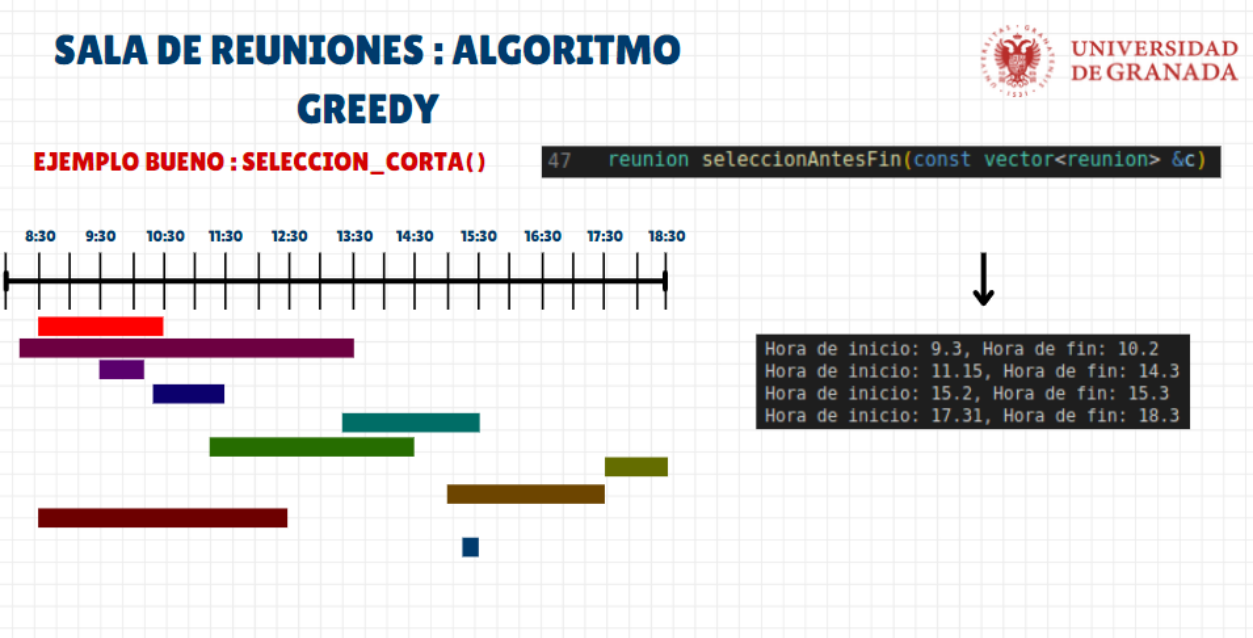
Está claro que con este ejemplo no se consigue la solución óptima seleccionando las reuniones que comiencen antes.

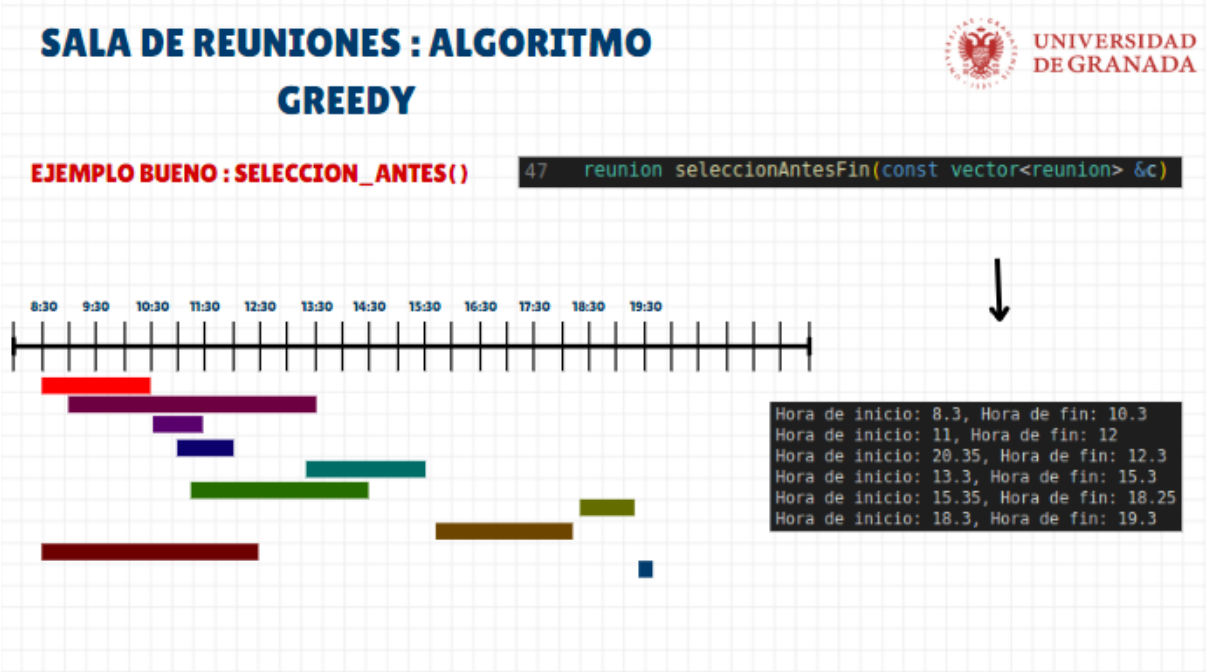
Probemos otro ejemplo que beneficia el ir seleccionando las reuniones más cortas:



Aquí se consigue la solución más óptima, pero ¿qué pasaría si la reunión más corta se solapara con las demás? arrojaría otra solución lejana de ser la óptima.

Como vimos en teoría la función que devuelve la solución óptima siempre es la que selecciona las reuniones que antes terminan:





Vemos que para diferentes ejemplos que benefician a otras funciones selección para que arrojen la solución óptima, provee la solución más óptima siempre.

Aunque demostrar la optimalidad de este tipo de algoritmos no es fácil, y a veces imposible, **la síntesis de las demostraciones mediante ejemplos, y la "técnica de intercambio de argumentos"** hacen que se pueda demostrar en este caso.

Técnica de intercambio de argumentos: se utiliza para demostrar que cualquier solución óptima puede ser transformada en una solución que es igual o peor que la solución proporcionada por el algoritmo voraz. Si se puede demostrar esto, entonces se puede concluir que la solución proporcionada por el algoritmo voraz es también óptima.

Supongamos que tenemos una solución óptima A que no coincide con la solución proporcionada por el algoritmo voraz B. Sea i la primera actividad en la solución A que no se encuentra en la solución B. Como la solución B comienza con la actividad que termina primero, debe haber una actividad j en B que termina antes que i. Podemos intercambiar las actividades i y j en la solución A sin violar las restricciones del problema, lo que resulta en una nueva solución A' que tiene al menos tantas actividades como la solución A original. Pero esto contradice la suposición de que A es óptima, porque si A' tiene al menos tantas actividades como A, entonces A' es también óptima, y esto significa que B no es óptima, ya que A' es igual o peor que B. Por lo tanto, se llega a la conclusión de que la solución proporcionada por el algoritmo voraz B es también óptima.

En resumen, la técnica de intercambio de argumentos se utiliza para demostrar que cualquier solución óptima puede ser transformada en una solución que es igual o peor que la solución proporcionada por el algoritmo voraz. Si se puede demostrar esto, entonces se

puede concluir que la solución proporcionada por el algoritmo voraz es también óptima. En el ejemplo del problema de selección de actividades, se utiliza esta técnica para demostrar que el algoritmo voraz que selecciona siempre la actividad que termina primero proporciona siempre la solución óptima.