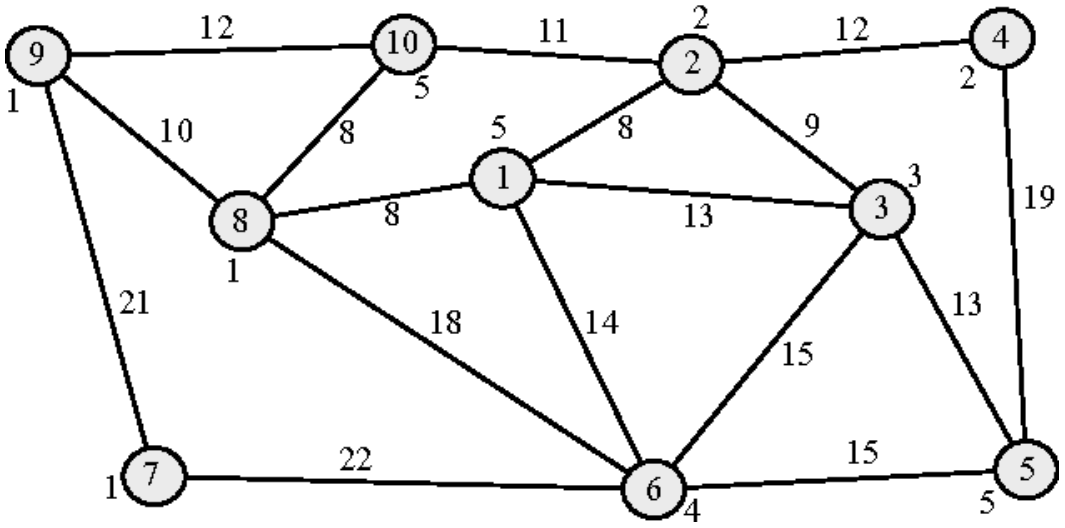
Práctica 4: Programación Dinámica

Defensa Grupo 3

- Feixas Galdeano, José Miguel
- Benaisa Cruz, Hamed Ignacio
- Espínola Pérez, Sergio
- Ruiz de Valdivia Torres, David Jesús

Problema a resolver

Se nos pide resolver el problema del viajante del comercio de nuevo, pero esta vez mediante programación dinámica. De esta forma podremos establecer una comparación entre este método y los algoritmos voraces de la práctica anterior.



Nuestra propuesta

Nuestra implementación en esencia está basada en el algoritmo de Held–Karp.

Básicamente se divide en dos partes:

- En la primera calculamos un primer camino con su primera distancia para establecerla como mínimo en primera instancia usando una función recursiva.

```
77
78  pair<double, list<int>> Grafo::distanciaConjunto(int indice, list<int> conjunto) {
79      list<int> aux = conjunto;
80      list<int> camino;
81
82      pair<double, list<int>> min, min_aux;
83      camino.push_back(indice);
84
85      if (conjunto.size() == 0)
86      {
87          return make_pair(matrizAdyacencia[indice-1][0],camino);   //Caso base, devolvemos la distancia entre el valor de indice y el primer elemento y el camino
88      }
89      else
90      {
91
92          list<int> camino_siguiente = camino;
93          int indice_siguiente = aux.front();
94          aux.pop_front();
95          pair<double, list<int>> resultadoHijo = distanciaConjunto(indice_siguiente,aux);
96          camino_siguiente.splice(camino_siguiente.end(), resultadoHijo.second);
97          min = make_pair(matrizAdyacencia[indice-1][indice_siguiente-1] + resultadoHijo.first, camino_siguiente);
98          aux.push_front(indice_siguiente);
99
100         list<int>::iterator it = aux.begin();
```

Nuestra propuesta (II)

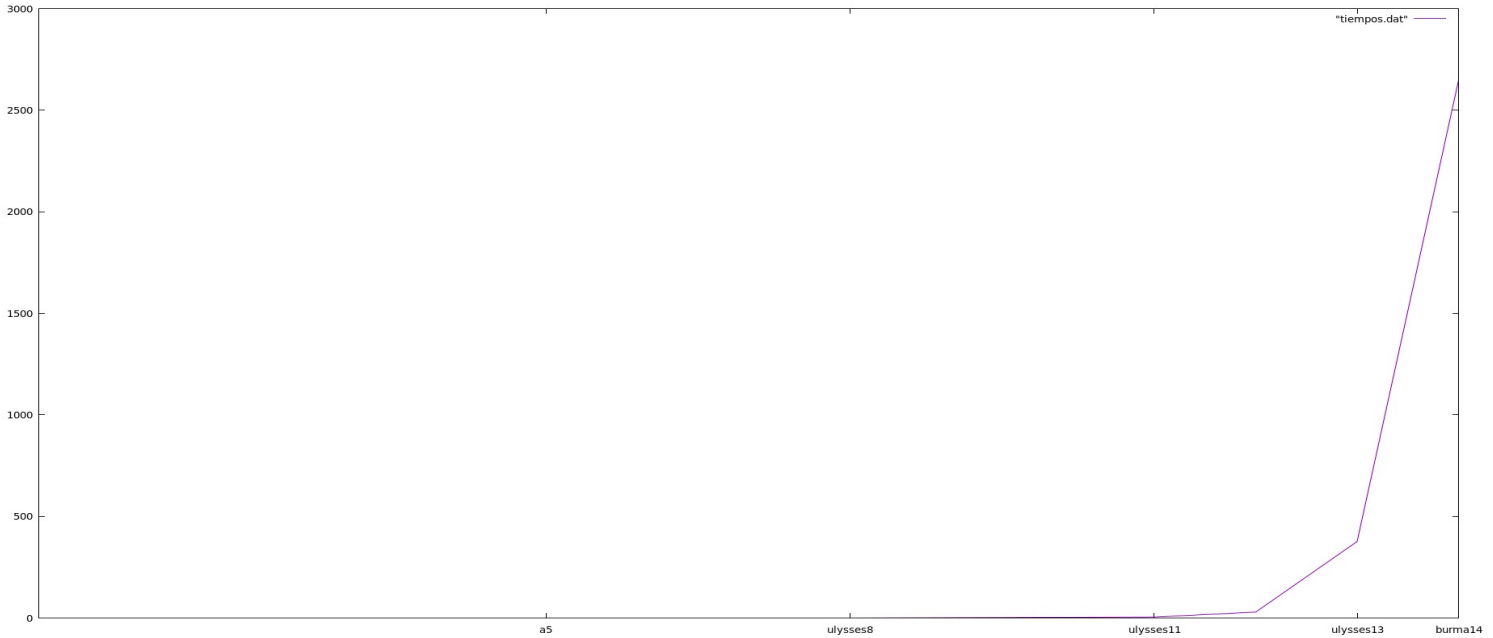
- Y en la segunda usamos la misma estructura recursiva para ir sacando los caminos y sus distancias e ir comparándolos con lo que sea nuestro mínimo en ese momento.

Finalmente, devolvemos el mínimo.

```
103         for (; it != aux.end(); ++it)
104         {
105             indice_siguiente = (*it);
106             it = aux.erase(it);
107 
108             
109             list<int> camino_aux = camino;
110 
111             pair<double, list<int>> resultadoHijo = distanciaConjunto(indice_siguiente,aux);
112             camino_aux.splice(camino_aux.end(), resultadoHijo.second);
113             min_aux = make_pair(matrizAdyacencia[indice-1][indice_siguiente-1] + resultadoHijo.first, camino_aux);
114             aux.insert(it,indice_siguiente);
115 
116             if(min_aux.first < min.first) {
117                 min = min_aux;
118             }
119         }
120 
121         return min;
122     }
123 }
```

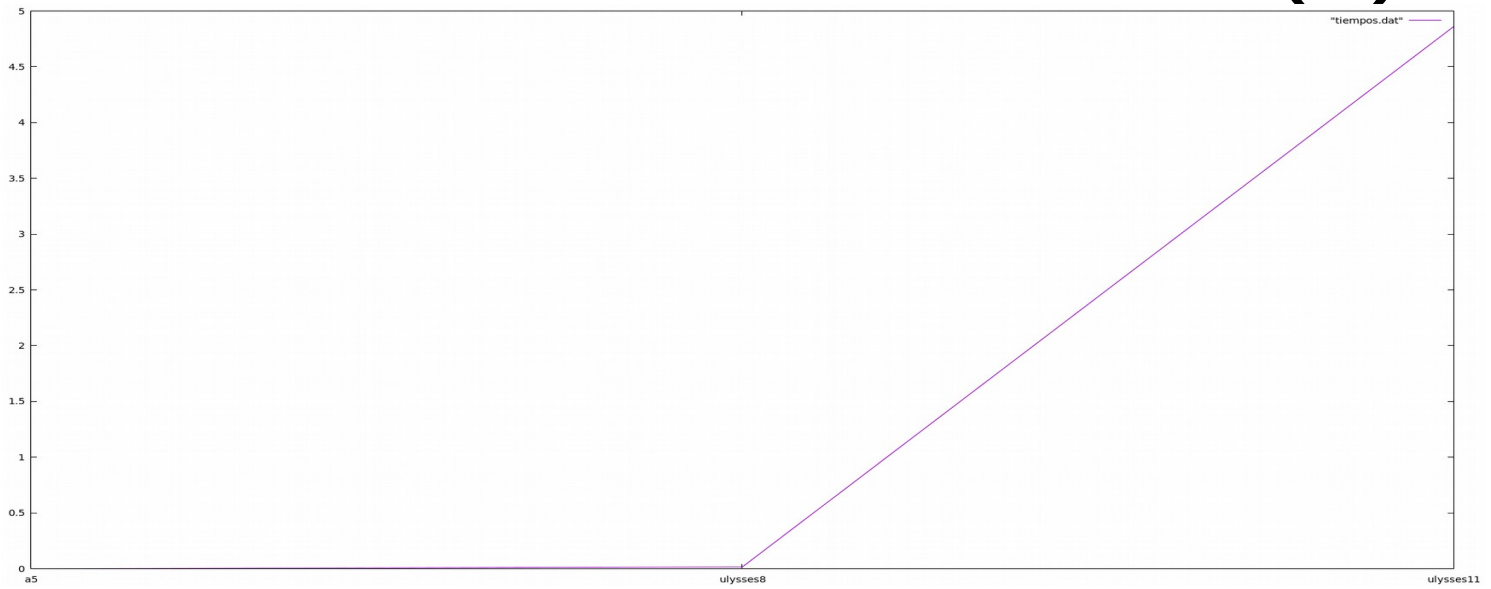
Resultados obtenidos

Se ejecutó el programa con TSP de distinto tamaño y se elaboró una gráfica con el resultado:



Se necesita ampliar la primera parte de la gráfica debido al gran incremento del tiempo.

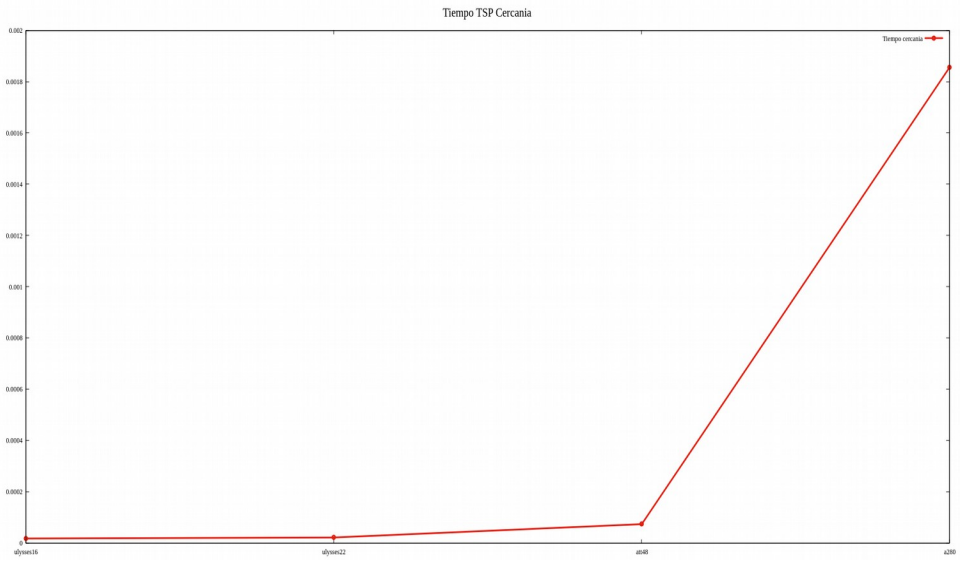
Resultados obtenidos (II)

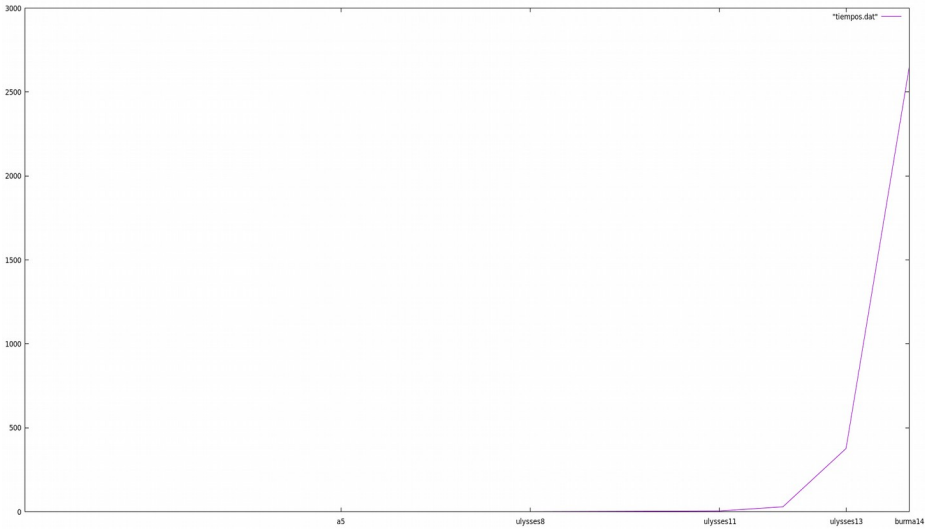


Podemos observar que el incremento de tiempo respecto al del tamaño es muchísimo mayor.

Comparación con algoritmos greedy

Podemos observar un grandísimo cambio pese a que el algoritmo greedy (por cercanía) maneja tamaños mucho mayores.





Comparación con algoritmos greedy

TSP	Greedy (distancia)	Programación dinámica (distancia)	Diferencia (%)	Tiempo greedy (s)	Tiempo programación dinámica (s)
a5	103,548	103,548	0,00 %	1,1E-05	0,000575576
ulysses8	38,484	37,8274	1,71 %	1,5E-05	0,0153359
ulysses11	78,969	70,3005	10,98 %	1,3E-05	4,86171
att12	23465,938	21179,4	9,74 %	1,6E-05	29,5285
ulysses13	80,611	70,6667	12,34 %	1,7E-05	376,914
burma14	38,688	31,3712	18,91 %	1,6E-05	2647,07

Conclusión

Como ya se venía a saber, el algoritmo voraz es mucho más eficiente pero no siempre da la solución óptima, por otra parte la solución basada en programación dinámica arroja la solución más óptima pero tardando mucho más tiempo.

UNIVERSIDAD DE GRANADA