1 TSP: Definició

Donada una col·lecció de ciutats, de les quals sabem el cost que implica viatjar d'una a qualsevol altra, el *Travelling Salesman Problem* (TSP) és trobar el camí de cost mínim que passi per cadascuna de les ciutats i retorni al punt d'origen. Més formalment, donat un graf complet no dirigit $G = (V, E)$ amb pesos $c(e) \geq 0$, hem de trobar el camí Hamiltonià de cost mínim.

Se sap que aquest problema és NP-complet, per la qual cosa avui en dia es desconeix una manera de resoldre una instància genèrica en temps polinòmic. En aquest document s'expliquen mètodes per trobar soluciones sub-òptimes mitjançant aproximacions i tècniques de cerca local.

2 TSP: 2-Aproximació

Els algorismes d'aproximació son algorismes que troben solucions sub-òptimes per a problemes d'optimització. L'avantatge d'aquests algorismes respecte d'altres, com per exemple la cerca heurística, és que podem donar cotes sobre el cost de la nostra solució respecte de l'òptima. Una 2-Aproximació és una solució que és, com a molt, dues vegades pitjor que la solució òptima.

Està demostrat que no existeix un algorisme d'aproximació per a una instància genèrica de TSP. No obstant, si afegim la restricció de que es compleixi la desigualtat triangular entre totes les ciutats (o sigui $\forall A, B, C\ distancia(A, B) \leq distancia(A, C) + distancia(C, B)$), el següent algorisme troba una solució per a TSP que és com a molt dues vegades pitjor que l'òptim:

- Obtenim un arbre mínim d'expansió T (*Minimum Spanning Tree* o MST) al nostre graf d'entrada G. Per fer-ho podem utilitzar l'algorisme de Kruskal o l'algorisme de Prim, que ben implementats poden funcionar en temps $\Theta(|E|log(|E|))$.
- Construïm el graf dirigit T' reemplaçant cada aresta de T per dos arcs de direccions oposades.
- Obtenim C, un cicle eulerià en T'. C és un cicle a G, en C apareixen tots els nodes però alguns estan repetits.
- Transformem C en un cicle hamiltonià C' eliminant les repeticions.
- C' és la nostra solució.

Demostrem ara que el nostre algorisme és una 2-aproximació. Per fer-ho hem de demostrar

que es dona la següent condició:

$$\frac{1}{2}A(x) \leq opt(x) \leq 2 * A(x)$$

Nota: Denotarem el cost d'un graf G com c(G), que vindrà definit com la suma dels pesos de les seves arestes.

Donat que TSP busca un mínim, sabem que la condició $opt(x) \leq 2 * A(x)$ sempre es compleix, per tant només hem de demostrar $\frac{1}{2}A(x) \leq opt(x)$. Per fer-ho, primer observem que, per definició de MSP, T és el conjunt d'arestes que uneixen tots els nodes de G', de pes mínim. Si retirem una aresta de la solució de TSP obtenim un Spanning Tree que podria ser o no el de mínim pes, però aquest fet ens diu que $c(T) \leq c(TSP)$.

El següent pas es demostra que $c(C') \leq c(C)$, això es únicament cert quan en G' es compleix la desigualtat triangular. Si tenim $C = \{v_1, v_2, ..., v_i, v_j, v_i, v_k, ..., v_n, v_1\}$, al computar $C' = \{v_1, v_2, ..., v_i, v_j, v_k, ..., v_n, v_1\}$, eliminant el repetit v_i, el que estem fent es canviar les arestes (v_j, v_i) i (v_i, v_k) per l'aresta (v_j, v_k). Gràcies a la desigualtat triangular sabem $c(v_j, v_i) + c(v_i, v_k) \geq c(v_j, v_k)$, per tant es compleix $c(C') \leq c(C)$.

Finalment, si ho posem tot a lloc obtenim:

$$c(C') \leq c(C) = 2 * c(T) \leq 2 * opt(x) \rightarrow \frac{1}{2}c(C') \leq opt(x)$$

2.1 Algorisme de Kruskal

Donat un graf connex no dirigit i ponderat, un arbre d'expansió d'aquest graf és un subconjunt d'arestes d'aquest que formen un arbre que uneix tots els vèrtexs. Més formalment, donat $G = (V, E)$ trobar un arbre d'expansió es trobar un subgraf $T = (V, A), A \subseteq E$ tal que T és un arbre, i trobar un arbre mínim d'expansió es trobar:

$$T^* = (V, A^*) \quad subjecte\ a \quad \arg\min_{A^*} \sum_{e \in A^*} c(e)$$

.

L'algorisme de Kruskal és un algorisme voraç que troba un arbre d'expansió mínim en un graf connex no dirigit i ponderat. Aquest ordena totes les arestes del graf de menor pes a major pes i, a continuació va creant l'arbre d'expansió T, afegint ordenadament a T aquelles arestes que no formin cicles a T. A continuació es mostra el pseudo-codi de l'algorisme:

Listing 1: Algorisme de Kruskal

```
Entrada: G = (V,E)

Ordenar E per ordre creixent
T := ∅
mentre |T| ≠ n − 1 fer
    e := la primera aresta de la llista ordenada
    treure e de la llista ordenada
    si e no genera un cicle en T llavors T = T ⋃ {e}
fi mentre

Sortida: T
```

2.2 Kruskal Eficient: Merge Find Set

Si analitzem el cost de l'algorisme de Kruskal, primer tenim $\Theta(|E|log|E|)$ degut a la ordenació de les arestes més un cost $\Theta(|E|)$ multiplicat per el cost de comprovar si una aresta genera cicle. En aquest apartat explicarem com fer aquesta comprovació de forma eficient.

Un Merge Find Set és una estructura de dades que realitza un seguiment d'un conjunt d'elements dividit en diferents subconjunts disjunts. Aquest tipus d'estructura suporta dues operacions bàsiques: *Buscar* és la operació que determina a quin subconjunt pertany un element en particular i es pot fer servir per saber si dos elements pertanyen al mateix conjunt; *Unir* uneix dos subconjunts en un de sol.

Durant la inicialització de l'algorisme de Kruskal col·locarem cada vèrtex del graf d'entrada en un subconjunt diferent i cada vegada que afegim una aresta al Spanning Tree unirem el dos conjunts als quals pertanyen els dos vèrtexs, que formen l'aresta. Si els dos vèrtexs d'una aresta pertanyen al mateix conjunt, descartarem aquesta aresta ja que afegir-la generaria un cicle en l'arbre d'expansió.

Per implementar aquesta estructura es pot fer servir un array d'enters (o un vector) on cada element farà referencia a un dels vèrtexs. Inicialitzarem l'array amb valors -1, ja que definirem que el fet de que en la posició x de l'array hi hagi un -1 indicarà que l'element x pertany al conjunt x. El fet de que en la posició x de l'array hi hagi un valor y, $y \neq -1$ ens indicarà que l'element x pertany al mateix conjunt que l'element y. Tenint en compte això a continuació mostrem el pseudo-codi de les operacions *buscar* i *unir*:

Listing 2: Buscar

```
Entrada: referenciaElement: enter, MFS: array d'enter

si (MFS[referenciaElement] == −1)
    retorna referenciaElement
```

```
sino
    retorna Buscar(MFS[referenciaElement],MFS)
fsi
```

Listing 3: Unir

```
Entrada: referenciaElementA: enter, referenciaElementB: enter,
MFS: array d'enter

conjuntA = Buscar(referenciaElementA,MFS)
conjuntB = Buscar(referenciaElementB,MFS)

si (conjuntA != conjuntB)
    MFS[conjuntA] := MFS[conjuntB]
    retorna cert
sino
    retorna fals
fsi
```

Per a millorar la eficiència de la funció *buscar* s'utilitzen les tècniques de compressió de camins i d'unió per talles. Ens podem imaginar els nostres conjunts com a elements que pengen, recursivament, d'altres elements, o sigui, com un arbre. Aquestes tècniques intenten fer que aquest "arbre" sigui lo més pla possible, reduint la profunditat dels camins i fent que siguin més amples.

La tècnica de compressió de camins intenta que tots els elements d'un conjunt apuntin al pare d'aquest. Això es pot fer modificant la funció *Buscar* per tal de que, en el moment que es torna de la recursivitat, s'assigni a *MFS*[*referenciaElement*] la referència al pare del conjunt.

Listing 4: Buscar

```
Entrada: referenciaElement: enter, MFS: array d'enter

si (MFS[referenciaElement] == -1)
    retorna referenciaElement
sino
    pare := Buscar(MFS[referenciaElement],MFS)
    MFS[referenciaElement] := pare
    retorna pare
fsi
```

Observem que per unir dos conjunts pengem el pare del conjunt B del pare del conjunt A, fent que tots els elements de B guanyin un grau de profunditat. Ens interessarà que el menor nombre d'elements possibles guanyin profunditat, així que la tècnica d'unió per talles consistirà en modificar la funció *Unir* per tal de penjar el conjunt petit del gran. Per controlar

el nombre d'elements d'un conjunt guardarem en MFS[x] (si x és el pare d'un conjunt) el nombre d'elements del conjunt en negatiu.

Listing 5: Unir

```
Entrada: referenciaElementA: enter, referenciaElementB: enter,
MFS: array d'enter

conjuntA = Buscar(referenciaElementA,MFS)
conjuntB = Buscar(referenciaElementB,MFS)

si (conjuntA != conjuntB)
    si (MFS[conjuntA] <= MFS[conjuntB])
        MFS[conjuntA] := MFS[conjuntA] + MFS[conjuntB]
        MFS[conjuntB] := conjuntA
    sino
        MFS[conjuntB] := MFS[conjuntA] + MFS[conjuntB]
        MFS[conjuntA] := conjuntB
    fsi
    retorna cert
sino
    retorna fals
fsi
```

Finalment, utilitzant un *Merge Find Set* l'algorisme de Krúskal quedarà com:

Listing 6: Algorisme de Kruskal

```
Entrada: G = (V,E)

Ordenar E per ordre creixent
T := ∅
MFS := Merge Find Set de mida |V|
mentre |T| ≠ n − 1 fer
    (u,v) = la primera aresta de la llista ordenada
    treure (u,v) de la llista ordenada
    si Unir(u,v,MFS) llavors
        T = T ⋃ {e}
    fsi
fi mentre

Sortida: T
```

El cost de la funció *unir* és, pràcticament, constant i el cost d'ordenar les arestes és $\Theta(|E|log(|E|))$, per tant, l'algorisme de Kruskal tindrà cost $\Theta(|E|log(|E|)) + \Theta(|E| * 1) = \Theta(|E|log(|E|))$.

2.3 Eliminació de Repetits

Com hem vist abans, C' és un cicle eulerià ja que en ell apareixen tots els nodes i alguns estan repetits. Resulta que eliminar aquests repetits segons un criteri o un altre pot donar lloc cicles hamiltonians diferents, amb costs diferents. Sabem que el cost d'aquestes solucions sempre serà com a molt dues vegades l'òptim, però ens interessa tenir la solució amb el menor cost possible.

Fer una cerca exhaustiva per tal de buscar la millor manera d'eliminar els nodes repetits comportaria temps exponencial. L'altre opció és buscar alguna estratègia més o menys intel·ligent per fer-ho. A continuació se'n proposen algunes:

- La manera més simple de fer aquest procés és, començant en algun punt del cicle, anar recorrent un a un tots els nodes i eliminar aquells que ens trobem per segona vegada.
- Un mica més elaborat seria provar la estratègia anterior començant des de cadascun dels nodes del cicle eulerià. D'aquesta manera obtindríem diversos cicles hamiltonians i ens quedaríem amb el de cost menor.
- Més elaborat encara seria dur a terme varis passos de simplificació. En cada pas buscaríem aquell conjunt de nodes repetits consecutius que, al eliminar-nos, donessin el cicle amb menor cost. Repetiríem aquest procés fins que no quedés cap node repetit.

3 TSP: Cerca Local amb Hill Climbing

Els mecanismes de cerca local consisteixen en a partir d'una solució inicial, per a un problema, explorar l'espai de solucions factibles a la cerca de solucions més bones. Per a això, es requereix una funció que, fent petites modificacions sobre una solució donada, et digui a quin conjunt de solucions es pot accedir. A aquest conjunt de solucions l'anomenem veïnat (solucions a explorar).

Un dels algorismes més simples és el de *Hill Climing*, que consisteix en, donada un solució S, obtenir el seu veïnat V i mirar quina de les solucions en V és la millor. Si la millor solució de $S' \in V$ és millor que S, repetim el mateix procés per a S', sinó, hem acabat i retornem S. A continuació hi ha el pseudo-codi:

Listing 7: Hill Climbing

```
S := GeneraSolucioInicial()
boolean fi := fals
mentre no fi
    V := obteVeinat(S)
    S' := millorSolucio(V)
```

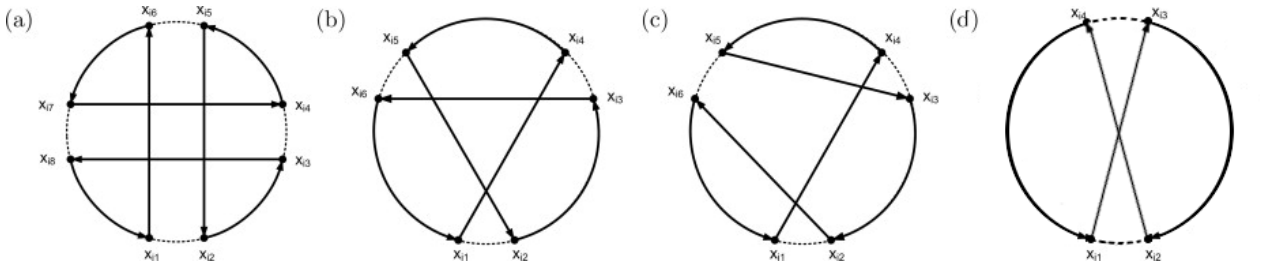


Figure 1: a) Double-Bridge b) 3-Opt versió 1 c) 3-Opt versió 2 d) 2-Opt

```
    si (esMillor(S',S))
        S = S'
    sino
        fi = cert
    fsi
fmentre
```

Hi han moltes funcions diferents per a calcular el veïnat d'una solució, a continuació es mostraran tres de les més típiques per al problema TSP (Figura 1):

- 2-Opt: L'objectiu d'aquesta funció es, donada una ruta que es creua per sobre de si mateixa, reordenar els nodes de la ruta de tal manera que això no succeeixi. Per a fer-ho, s'agafa un fragment del cicle hamiltonià que forma la solució d'un TSP i s'inverteix l'ordre en que es visiten els nodes (Figura 2). Un cerca local completa compararà totes les possibles modificacions que es puguin fer mitjançant aquest mecanisme, sobre una solució donada.
- 3-Opt i Double-Bridge són funcions similars a la 2-Opt. En aquest cas es modifiquen, respectivament, dos o tres fragments del cicle.

Podem agafar la aproximació explicada anteriorment com a solució inicial per a l'algorisme de *Hill Climbing* i millorar a partir d'aquí. D'aquesta manera coneixerem una cota de com de bo és la nostra solució però segurament acabarem tenint alguna cosa força millor. El fet començar amb una solució bona farà que *Hill Climbing* convergeixi més ràpid i, intuïtivament, també ens portarà a solucions millors.

4 TSP: Cerca Local Iterada

El major problema de la cerca local és que convergeix ràpidament cap a un mínim local i, un cop hi arriba, no pot escapar d'aquest. Altres esquemes com la cerca local iterada (*Iterated Local Search*, ILS) intenten resoldre aquest problema.

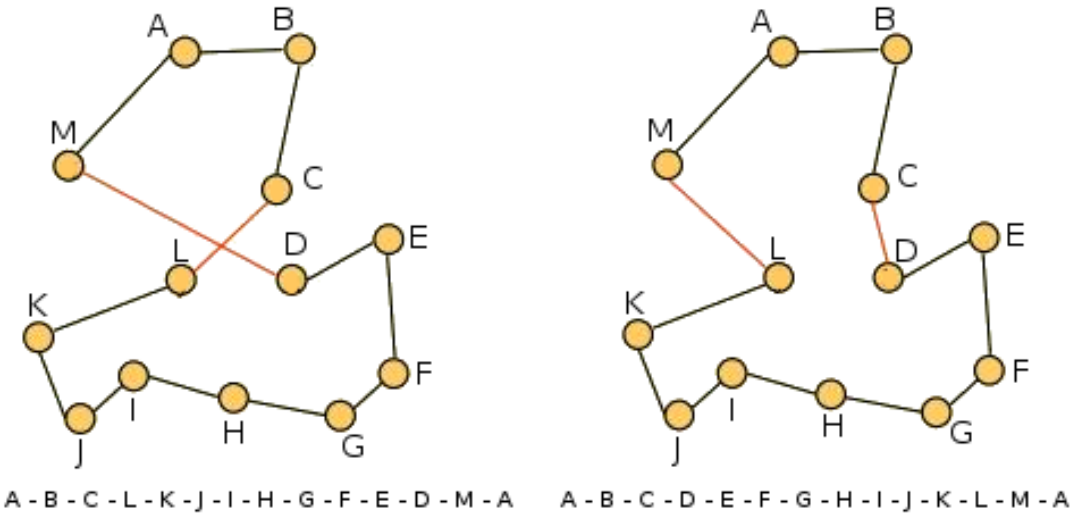
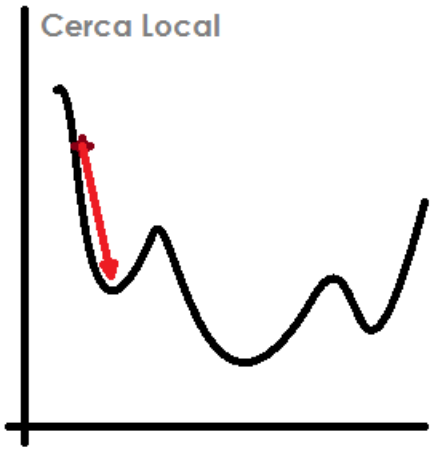
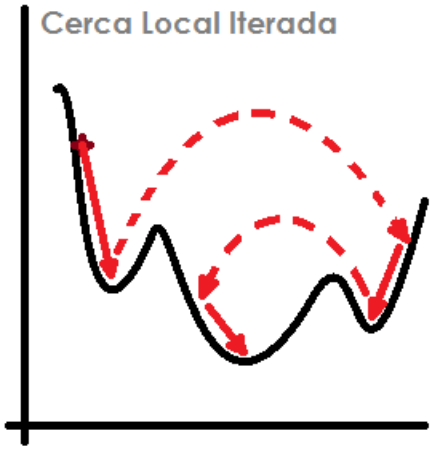


Figure 2: exemple 2-Opt



Un esquema de cerca local iterada consisteix en, a partir d'una solució inicial S, realitzar una cerca local sobre S fins que convergeixi en una solució S', a continuació aplicar una pertorbació a S' per obtenir S'' i repetir el mateix procés per a S''. En tot moment ens guardarem la millor solució trobada fins al moment i hem de fixar el temps que deixarem que l'algorisme busqui.

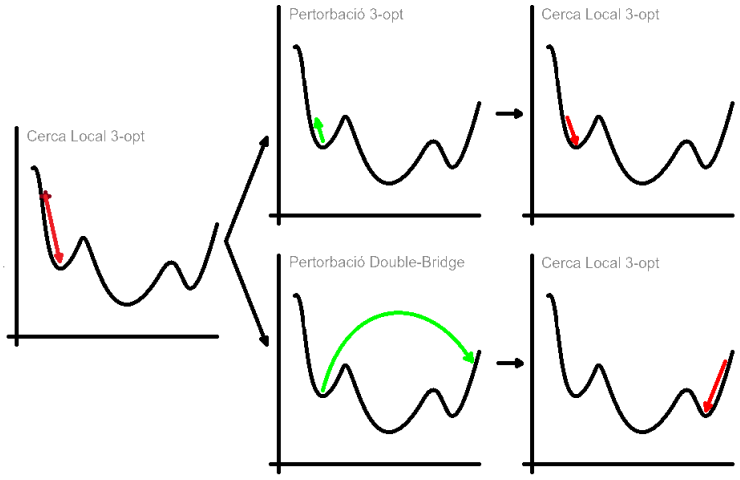


Hi han varies variants d'aquest algorisme: podem, per exemple, decidir que sempre pertorbarem la millor solució trobada fins al moment (estratègia d'intensificació) o fer tot el contrari i pertorbar sempre la última solució trobada (estratègia de dispersió). D'això li direm funció d'acceptació. El pseudo-codi de l'algorisme seria el següent:

Listing 8: Iterated Local Search

```
S := GeneraSolucioInicial()
S* := S
mentre quedi temps fer
    S' := CercaLocal(S)
    S*:=optim(S*,S')
    S':=acceptacio(S*,S')
    S''=pertorbacio(S')
    S := S''
fmentre
```

és important que el canvi produït per la funció de pertorbació no pugui ser desfet fàcilment per la cerca local, per exemple, si per la cerca local utilitzem el veïnat 3-opt i la funció de pertorbació transforma S en una solució $S' \in 3 - opt(S)$, acabarem sempre en el mateix mínim local. En aquest cas, per solucionar aquest problema, podríem fer que definir una funció que transformes S en $S' \in double - bridge(S)$.



Altre cop, podríem utilitzar l'aproximació com a solució inicial per al nostre algorisme.