Solucions Paral·leles del TSP utilitzant Oracle Coherence

Albert Palau
Grau en Enginyeria Informàtica
Especialitat en Computació
Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Director : Pablo Estrada Ortega
Empresa : Oracle
Ponent : Ricard Gavaldà Mestre
Departament : LSI

18 de juny de 2013

Contents

I Contextualització i planificació del treball **5**

1 Definició del problema **6**
1.1 Context 6
1.1.1 Traveling Salesperson Problem 7
1.1.2 Mètodes per resoldre el TSP 7
1.1.3 Oracle Coherence i processament paral·lel 9
1.1.4 Agents Involucrats 14
1.1.5 Estat de l'art 15

2 Planificació i gestió del treball **16**
2.1 Abast del projecte 16
2.1.1 Objectius 16
2.2 Requisits 17
2.3 Recursos 18
2.4 Planificació 19
2.4.1 Descomposició de processos en tasques 19
2.4.2 Gantt del projecte 20
2.4.3 Valoracions d'alternatives i pla d'accions 20
2.5 Pressupost 22
2.5.1 Altres costos associat al projecte 24
2.6 Sostenibilitat i compromís social 24
2.7 Lleis i regulacions 25
2.7.1 Documentació i bibliografia 25
2.7.2 Software 25

II Cos del treball **27**

3 Solucions seqüencials **28**
3.1 Resolució mitjançant un arbre d'expansió mínim 28
3.1.1 Rati dels algorismes d'aproximació 29
3.1.2 Arbre d'expansió mínim com a resolució del TSP 29
3.2 Resolució mitjançant l'algorisme Lin-Kernighan Helsgaun 32
3.2.1 Descripció 32

3.2.2 Lin-Kernighan 33
3.2.3 Lin-Kernighan Helsgaun 38
3.3 Resolució mitjançant l'algorisme Ant Colony 43
3.3.1 Descripció 43
3.3.2 Modelització del ACO per al TSP 45
3.4 Resultats obtinguts 48
3.4.1 Comparativa 48

4 Implementacions paral·leles 50
4.1 Implementació Paral·lela Lin-Kernighan-Helsgaun 50
4.2 Implementació Paral·lela Ant Colony 51
4.3 Resultats obtinguts 53
4.3.1 1-core 53
4.3.2 2-core 54
4.3.3 3-core 54
4.3.4 4-core 55

5 Conclusions 56
5.1 Revisió dels objectius i requisits 56
5.2 Treball futur 57

III Bibliografia 58

IV Annexos 61

6 Funcions Java 62
6.1 Lin-Kernighan Helsgaun 62
6.1.1 Càlcul de $\beta(i, j)$ 62
6.1.2 Càlcul subgradient 63
6.1.3 Tour inicial 65
6.1.4 Determinar si un r-move genera un tour 67
6.2 Ant colony 68
6.2.1 Sel·lecció de la següent ciutat (roulette wheel selection) . 68
6.2.2 Actualització de les feromones 69
6.3 Versió paral·lela Lin-Kernighan Helsgaun 69
6.3.1 Esquelet principal 69

Abstract

The main objective of this project is to solve in different ways the traveling salesperson problem, one of the most famous combinatorial problems. The best options to solve it are going to be implemented and parallelized with the Oracle Coherence framework to show the results and extract conclusions. To do that, we shall use a benchmark data set called TSPLib which is one of the most tested dataset in this area. At the end we shall review the results to help for incoming people who want to solve this problem.

Resumen

El objetivo del presente trabajo es resolver con diferentes alternativas el famoso problema combinatorio del viajante de comercio. Las mejores alternativas encontradas serán implementadas y paral·lelizadas con el framework de Oracle Coherence para mostrar los resultados y poder extraer conclusiones. Para ello se utiliza un conjunto de datos de benchmark llamado TSPLib que ha sido ampliamente probado por un gran número de expertos. Al final del trabajo se valoraran los resultados obtenidos para que cualquier interesado en resolver instancias del mismo problema tenga conocimiento de las diferentes maneras de resolverlo y/o vea cual de ellas se adapta mejor a sus necesidades.

Resum

L'objectiu d'aquest treball és resoldre de diferents maneres el famós problema combinatori del viatjant de comerç. Les millors alternatives trobades seran implementades i paral·lelitzades amb el framework d'Oracle Coherence per mostrar els resultats i poder extreure conclusions. Per fer això utilitzarem un conjunt de dades benchmark anomenat TSPLib el qual ha sigut àmpliament testejat per un gran número d'experts. Al final d'aquest reball es valoraran els resultats obtinguts perquè qualsevol interessat en resoldre instàncies del mateix problema tingui un bagatge més ampli sobre la matèria i/o per escollir quina de les solucions aquí exposades s'adapta millor a les seves necessitats.

Part I

Contextualització i planificació del treball

Capítol 1

Definició del problema

1.1 Context

Aquest projecte es realitza en el marc d'un conveni de cooperació educativa amb l'empresa Oracle Iberica SRL. Oracle ha desenvolupat un framework anomenat Coherence [7], que entre altres característiques permet el processament paral·lel de dades. El projecte sorgeix de la possibilitat de crear una PoC per a un client que necessita resoldre instàncies del Traveling Salesperson Problem. D'aquesta manera es va consensuar amb el director del projecte i el ponent la factibilitat d'accelerar solucions TSP utilitzant el framework Coherence.

El TSP, que descriurem amb més detall a la Secció 1.1.1, és un problema combinatori de gran importància que pertany a la classe dels anomenats NP-complets [1]. Aquests tipus de problemes no es poden tractar sempre amb algorismes exhaustius ja que els recursos necessaris per la seva execució creix d'una forma exponencial amb la grandària de l'entrada. Això vol dir que amb problemes petits els recursos necessaris (temps, memòria, ...) són tan grans que no és viable resoldre'ls fins i tot fent ús de tècniques de paral·lelització, ja que, al créixer exponencialment, necessitaríem que els processadors també creixessin exponencialment amb la mida de l'entrada.

Donat que els algorismes exhaustius no sempre són una opció per resoldre el TSP (qüestions de temps, recursos, ...) hi ha altres alternatives que, tot i no garantir que troben la solució òptima, poden trobar solucions raonablement bones a la pràctica en un temps moderat. En aquest treball ens centrarem en aquests tipus d'algorismes i en trobar tècniques de paral·lelització utilitzant Coherence. Es tracta doncs d'aprofitar els processadors disponibles de la millor manera possible. En aquest punt mirarem l'speedup [1] i l'eficiència [2].

[1] Quant més ràpid va la versió paral·lela d'un algorisme que la seqüencial en p processadors $S_p = T_1/T_p$

[2] $E_p = S_p/p$

1.1.1 Traveling Salesperson Problem

Descripció

El problema del viatjant de comerç és un problema NP [1] molt estudiat des de fa molt de temps per la seva importància pràctica. El problema s'explica informalment pensant en un comerciant que, començant en una ciutat, vol trobar el camí més curt per passar per les ciutats dels clients i tornar a la ciutat d'inici visitant cada ciutat una única vegada (excepte la d'inici). Anomenarem aquest camí un *tour*.

Formalment, el TSP es pot representar com un graf complet $G = (N, A)$ on N és el conjunt de nodes que representen les ciutats i A és el conjunt d'arcs. Cada arc $(i, j) \in A$ té associat un cost d_{ij} positiu que reflecteix el cost d'anar de i a j, amb $i, j \in N$. El cost d'un camí és la suma dels costos de les arestes que el formen. Si la representació del nostre problema genera un graf que no és complet i, per exemple, per anar a un node i fins a un node j no hi ha una aresta directa, afegim una amb cost igual que el del camí mímin que va de i a j des de j. L'objectiu en el TSP és buscar un cicle Hamiltonià [3] o tour amb cost associat mínim. Així doncs una solució òptima del TSP serà una permutació π de nodes amb índex $\{1, 2, ..., n\}$ tal que minimitzi el cost associat a π, definit com:

$$f(\pi) = \sum_{i=1}^{n-1} d_{\pi(i)\pi(i+1)} + d_{\pi(n)\pi(1)} \tag{1.1}$$

En aquest treball, per simplificació, treballarem només amb els grafs tals que els costos de les seves arestes compleixen les següents propietats:

- $\forall i, j \in N, c_{ij} = c_{ji}$. Els grafs que compleixen aquesta propietat s'anomenen *simètrics*.
- $\forall i, j, k \in N, c_{ik} \leq c_{ij} + c_{jk}$. Aquesta inequació es diu inequació triangular i dona lloc a problemes *mètrics*.
- Si c_{ij} són distàncies euclidianes entre punts d'un pla, el problema s'anomena *Euclidià*. Un problema Euclidià és tan simètric com mètric.

Amb aquestes restriccions de les instàncies del problema, el TSP continua sent un problema NP-complet [9].

1.1.2 Mètodes per resoldre el TSP

Els algorismes per resoldre el TSP es poden dividir en dues classes:

- Algorismes exactes
- Algorismes heurístics i aproximats

[3] Un cicle Hamiltonià és un cicle d'un graf $G = (V, E)$ no dirigit que conté tots els vèrtex en V.

Algorismes exactes

Aquest tipus d'algorismes sovint troben la solució òptima en un número limitat de passos. Avui en dia amb la potència de les màquines que tenim podem trobar, mitjançant aquest tipus d'algorismes, solucions per a problemes relativament grans (milers de ciutats) [5, 6].

Els algorismes més efectius d'aquest tipus són els mètodes *cutting-plane* o *facet-finding* [4, 5, 6]. No obstant aquests algorismes queden fora de l'abast d'aquest treball per la seva complexitat.

Algorismes aproximats i heurístics

Aquest tipus d'algorismes obtenen a la pràctica bones solucions aproximades en un temps d'execució assumible. No obstant, poden tenir algunes instàncies difícils per a les quals o bé tarden molt de temps o bé donen solucions arbitràriament pitjor que l'òptim. En la pràctica aquest tipus de implementacions difereixen un cert factor del òptim a canvi de trobar solucions més eficientment.

Els algorismes aproximats tenen la característica que sempre pots indicar un factor tal que amb totes les instàncies la solució trobada per l'algorisme no seran pitjors que l'òptim pel factor corresponent. Això se'n diu *k-aproximat*, on la k és un factor tal que, si C^* és el cost del tour òptim i C és el cost de la solució donada per l'algorisme llavors $C/C^* \leq k$ per qualsevol instància del problema.

Pel que fa als heurístics no podem fitar la bondat de les solucions. Només ens podem basar en fets experimentals. De fet un heurístic pot anar molt bé en mitjana però pot existir casos patològics on l'algorisme és molt dolent.

Aquest tipus d'algorismes es poden classificar en tres classes diferents:

- Algorismes constructius
- Algorismes de millora
- Algorismes que utilitzen les dos metodologies anteriors

Els algorismes constructius creen un camí afegint nodes, fins haver-los afegit tots, moment en que tenen un tour. Els algorismes de millora comencen amb un tour inicial i l'intenta millorar a partir de canvis. Els compostos combinen els dos mètodes.

1.1.3 Oracle Coherence i processament paral·lel

Descripció

Oracle Coherence [7] és, essencialment, un *data grid* en memòria (o Cache). Es a dir, conté informació que podria estar en una base de dades carregada en memòria RAM convencional.

Coherence te moltes funcionalitats i característiques diferents. Per exemple:

- Una varietat d'estratègies de caching
- Diferents models de esdeveniments
- Diferents models de processament paral·lel
- ...

L'esquema general d'execució de Coherence pot veure's a la Figura 1.1

Primer de tot definirem alguns conceptes que són necessaris per entendre el producte.

- Node d'un cluster Coherence. És un procés Java que pertany a un Cluster o és una instància d'un servidor Coherence. Pot contenir dades, processos i reaccionar a diferents esdeveniments. Generalment té un fitxer de configuració de cache (.xml) que el defineix.
- Cluster. És un grup de nodes que poden guardar informació i que tenen un nom associat de cluster. Tots ells treballen junts per oferir el *data grid*.

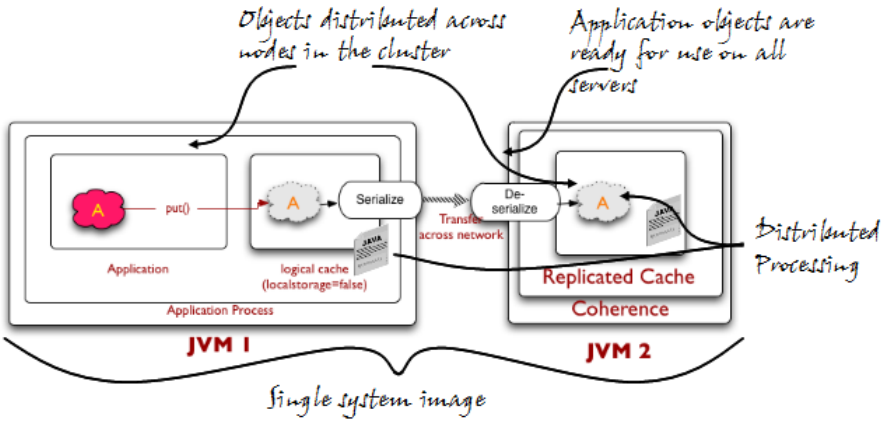

Figura 1.1: Entorn Coherence

Tipologies de Cache

En un cluster Coherence es poden definir diferents tipus de cache. Segons la seva tipologia aquesta es comportarà d'una manera o un altre. Les més habituals són:

- Cache local: Quan una aplicació java accedeix a una cache local la informació que hi ha i que pot modificar només afectarà a la JVM [4] on es troba. Un exemple de l'arquitectura d'aquesta tipologia el podeu trobar a la Figura 1.2

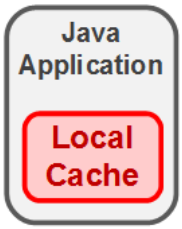

Figura 1.2: Cache local

- Cache replicada. Replica tota la informació que es modifica a tots els membres del cluster. Està dissenyada per un alt rendiment en la lectura. Ofereix una replicació transparent als diferents membres del cluster. Un exemple de l'arquitectura d'aquesta tipologia el podeu trobar a la Figura 1.3

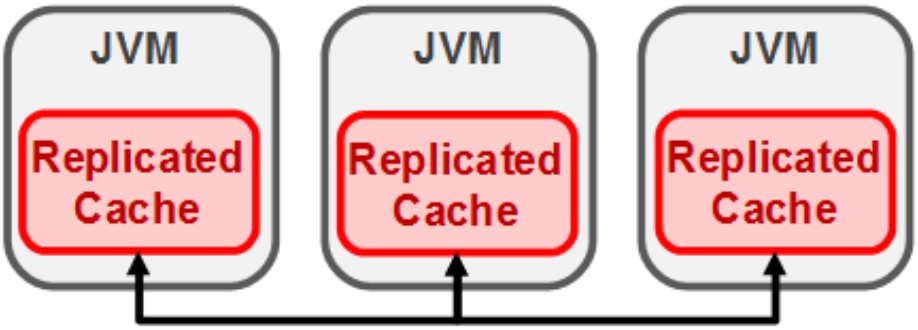

Figura 1.3: Cache replicada

- Cache particionada. És una cache que parteix la informació en una copia primaria i secundaria i les reparteix a través del cluster. Ofereix seguretat de les dades enfront fallides i proporciona una certa escalabilitat i eficiència. Un exemple de l'arquitectura d'aquesta tipologia el podeu trobar a la Figura 1.4

[4] Java Virtual Machine

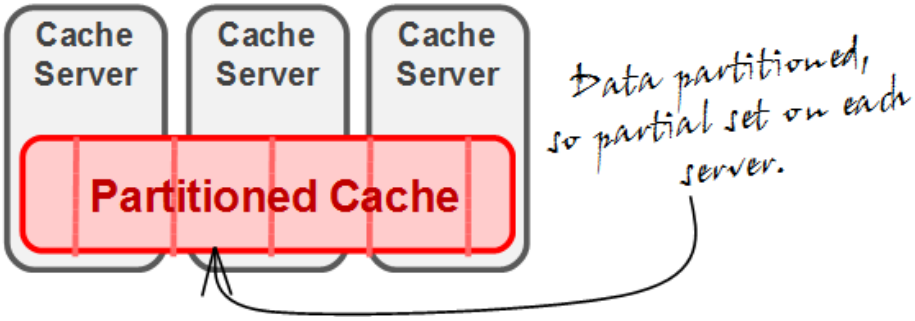

Figura 1.4: Cache particionada

- Cache *Near*. És una cache que combina una particionada anomenada *back tier* i una cache local com a *front tier*. Aquesta arquitectura permet tindre un accés local molt ràpid i una més lenta però amb més capacitat a la cache particionada. Un exemple de l'arquitectura d'aquesta tipologia el podeu trobar a la Figura 1.5

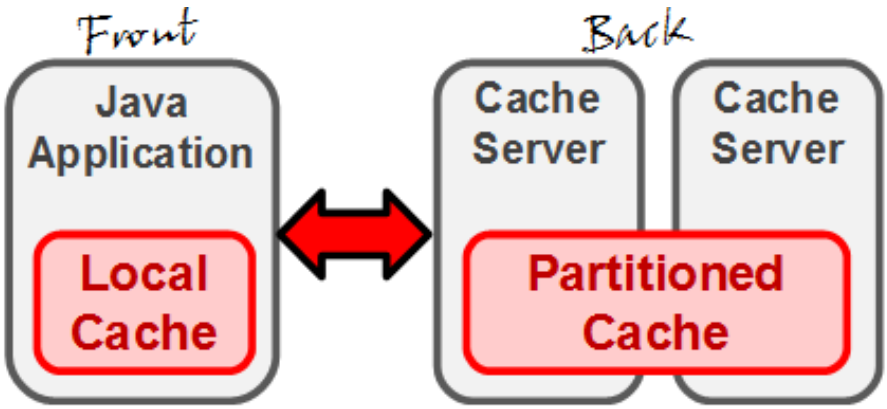

Figura 1.5: Near Cache

Processament paral·lel

Coherence ofereix dos tipus de model de processament: per esdeveniments i processament paral·lel.

El processament paral·lel inclou:

- *Entry Processors*: Equivalent a un agent que s'executa en paral·lel en les **dades** dels diferents membres del cluster.
- *Invocation Services*: És un agent que s'executa en els membres del cluster i que normalment no està relacionat amb les dades de les caches.

Respecte els esdeveniments hi ha:

- *Map Listeners*: Els quals s'executen **asíncronament després** d'un canvi en les dades.
- *Map triggers*: Els quals s'executen **síncronament abans** d'un canvi en les dades.

Per aquest treball utilitzarem només el processament paral·lel.

Consistència de les dades

Coherence ofereix diferents maneres de fer transaccions, cadascuna amb diferents nivells de garantia. Uns exemples són:

- Control mitjançant locks explicits (*ConcurrentMap*)
- Lliure de programar amb locks (*Entry Processors*)
- Mitjançant un framework de transaccions que ofereix Coherence.

Model de programació (treballant amb les caches)

Per poder treballar amb les caches hem de modelar les nostres dades/objectes amb estructures de clau/valor. Les caches no funcionen amb altres estructures de dades. Les operacions que es poden fer són del estil *get* i *put*. Funciona idènticament a com ho fa *java.util.Map* [5]

Named Caches

Les *Named Caches* són simplement *Maps*, es a dir, un conjunt de parells identificador/valor. Contenen la informació compartida per un cluster i localitzat a memòria. Generalment tenen un fitxer de configuració (.xml) per configurar les seves característiques. Un exemple de fitxer de configuració seria:

```
<cache-config>
<cache-scheme-mapping>
   <cache-mapping>
      <cache-name>example-cache</cache-name>
      <scheme-name>example-scheme</scheme-name
    </cache-mapping>
</cache-scheme-mapping>
...
  <cache-schemes>
  <distributed-scheme>
     <scheme-name>example-scheme</scheme-name>
     <service-name>DistributedCache</service-name>
     <backing-map-scheme>
       <local-scheme>
       </local-scheme>
      </backing-map-scheme>
      <autostart>true</autostart>
```

[5] http://docs.oracle.com/javase/6/docs/api/java/util/Map.html

```
    </distributed-scheme>
  </cache-schemes>
</cache-config>
```

on l'etiqueta *cache-scheme-mapping* defineix i **dona nom** a les caches, i l'etiqueta *cache-schemes* defineix la tipologia de cache.

Una vegada s'ha definit i configurat les caches, podem fer ús d'elles en les aplicacions. Un tros de codi bàsic podria ser:

```
import com.tangosol.net.CacheFactory;
import com.tangosol.net.NamedCache;

public class TestCache {

    public static void main(String[] args) {

        String key = "key";

        CacheFactory.ensureCluster();



        NamedCache cache=
                   CacheFactory.getCache("myCache");


        cache.put(key,value);
        Object o = cache.get(key);


       CacheFactory.shutdown();
    }
}
```

Executant *Invocation Services*

La forma de utilitzar un cluster coherence per fer execucions paral·leles, i que no siguin només operacions sobre dades, és a traves dels Invocation Services. Aquests serveis, com les caches, han de ser definides en el fitxer de configuració:

```
<caching-schemes>
  ...
  <invocation-scheme>
    <scheme-name>MyInvocationServiceScheme</scheme-name>
    <service-name>InvocationService</service-name>
    <thread-count>5</thread-count>
    <autostart>true</autostart>
  </invocation-scheme>
  ...
</caching-schemes>
```

Una vegada definit el servei es pot utilitzar en qualsevol aplicació java:

Listing 1.1: Execució paral·lela síncrona d'una classe Java

```
public class CheckNumberOfCPUsAgent extends AbstractInvocable {
  public void run() {
    setResult(Runtime.getRuntime().availableProcessors());
  }
}


InvocationService service = (InvocationService) CacheFactory.getService("
     InvocationService");

Map<Member, Integer> numCPUMap =
service.query(new CheckNumberOfCPUsAgent(), null);

for (Map.Entry<Member, Integer> numCPU : numCPUMap.entrySet()) {
  System.out.println("Member: " + numCPU.getKey() +
                     " Number of CPUs: " + numCPU.getValue());
```

La classe *AbstractInvocable* és una classe abstracta que implementa la interfície *Invocable* amb la implementació d'algunes funcions bàsiques. Aquestes classes són les que s'han de implementar per poder executar l'agent.

En el cas d'una execució asíncrona haurem de implementar una classe Observer que *observarà* si un membre falla en l'execució, acaba, s'uneix, etc.

Listing 1.2: Execució paral·lela asincrona d'una classe Java

```
private static class CPUInvocationObserver implements InvocationObserver {
  // Cridat quan cada membre completa asincronament l'execució de l'agent.
  public void memberCompleted(Member member, Object result) {
    System.out.println("Member: " + member + " Num CPUs: " + result);
  }
  // Cridat quan un membre falla duran l'execució asincrona de l'agent
  public void memberFailed(Member member, Throwable throwable) {}

  // Cridat quan un membre deixa el cluster quan està executant l'agent
  public void memberLeft(Member member) {}

  // Cridat quan tots els membres han acabat l'execució de l'agent
  public void invocationCompleted() {}
}

.......
//Main
InvocationService service = (InvocationService) CacheFactory.getService("
     InvocationService");

service.execute(new CheckNumberOfCPUsAgent(), null,
new CPUInvocationObserver());
}
```

1.1.4 Agents Involucrats

Són moltes àrees on la necessitat de resoldre instàncies del TSP de forma eficient és preferible o necesari. Algunes d'aquestes àrees poden ser [6]:

- Seqüenciació del Genoma Humà

[6] http://www.tsp.gatech.edu/apps/index.html

- *Starlight Interferometer Program* (Estalvi de combustible en missions espacials)
- Optimització de circuits
- Cadenes de DNA universals
- Optimitzar el camí d'un distribuidor en la ciutat. (logística)
- (...)

Qualsevol persona amb una necessitat de resoldre el TSP podria ser-li útil aquest treball de fi de grau on s'expliquen algunes tècniques i metodologies.

1.1.5 Estat de l'art

El TSP és probablement el problema més clàssic d'optimització discreta. Per aquest motiu hi ha gran interès en resoldre'l quan es treballa en aquest camp i per tan s'ha generat força informació al respecte.

L'any 1832 es va fer la primera descripció informal del problema. Més tard, en els anys 60, *Proctor and Gamble* van guanyar $10K en una competició resolent una instància del TSP de 33 ciutats. A partir d'aquesta data va créixer un interès particular pels models estadístics. Richard Karp, l'any 1972, va demostrar que el TSP és un problema NP-complet [14]. A partir d'aquest moment el desenvolupament es va dividir en dues branques:

- Algorismes exactes utilitzant programació lineal. Segons Keld Helsgaun [7] el problema més gran resolt fins l'any 2006 ha sigut de 85.900 ciutats.
- Heurístics. La instancia més gran resolta ha sigut de 1,904,711 ciutats i amb una aproximació del 0.056% del òptim, l'any 2009 [13].

Pel que fa als algorismes exactes els més efectius son els algorismes de *cutting-plane* o *facet-finding*. Són algorismes força complexos i sofisticats [4, 5, 6].

Respecte els heurístics l'algorisme més efectiu és la variació del Lin-Kernighan feta per Keld Helsgaun [13].

[7] `http://www.akira.ruc.dk/~keld/research/LKH/`

Capítol 2

Planificació i gestió del treball

2.1 Abast del projecte

2.1.1 Objectius

En aquest treball he hagut de prendre decisions que marcarien el treball posterior. Per exemple; la selecció dels algorismes a implementar per fer front al TSP i la estratègia de paral·lelització a aplicar.

El principal objectiu d'aquest treball es veure les diferents maneres de fer front a un problema NP-Complet com és el TSP. Per veure quina opció és millor i contrastar resultats en aquest treball implementarem tres algorismes :

- Construirem un arbre d'expansió mínim [1] del graf i després el recorrerem en preordre. Aquest algorisme és del tipus constructiu i *2-aproximat* (garantirem que la solució trobada no és mai pitjor que 2 vegades l'òptim)
- Ant Colony [2] és una metaheurístic [1]. Aquest algorisme es basa en el comportament de les formigues per trobar un camí mínim. És del tipus d'algorismes que milloren un tour inicial i no té cap garantia d'aproximació.
- Lin-Kernighan Helsgaun [3], és una optimització de l'algorisme Lin-Kernighan original. Es basa en el que s'anomena moviments *k-opt* i tracta de fer canvis sobre un tour inicial treien k links i afegint k nous links. Aquest algorisme al igual que l'Ant Colony no té cap garantia d'aproximació encara que es considera el millor a la pràctica.

[1] Una metaheurística és un mètode heurístic per a resoldre un tipus de problema computacional general, utilitzant els paràmetres donats per l'usuari sobre uns procediments genèrics i abstractes d'una manera que s'espera eficient. Normalment, aquests procediments són heurístics.

Aquestes implementacions intentaran ser paral·lelitzades amb *Coherence* i es comprovaran diferents propietats com: temps d'execució, aproximació a l'òptim, etc. Finalment amb aquests resultats farem una conclusió de les diferents maneres de resoldre el TSP.

Pel que fa a els algorismes, el que he volgut ha sigut agafar un conjunts ben diferents, assequibles d'implementar amb el temps disponible i que no tots siguin de la mateixa classe, es a dir: el que utilitza bàsicament el MST està classificat com un constructiu i és aproximat amb un factor de 2. L'Ant Colony és un metaheurístic que millora un tour inicial i Lin-Kernighan Helsgaun està considerat com uns dels millors algorismes per resoldre el TSP. En aquest també es veurà el concepte de *k-opt*. Aquesta tria ha sigut consensuada amb el director i el ponent.

Per altra banda he descartat els exhaustius per la seva complexitat i perquè, si agafava un d'aquests, no podia assegurar que podria implementar cap altre no exhaustiu.

2.2 Requisits

Ens plantegem una serie de requisits que el projecte ha de complir. Són els següents:

		Cicle de Vida			
		Planificació	Materialització	Utilització	Reintegració
Valor Projecte	Diners			L'ús de la solució ha de disminuir la despesa original per resoldre aquest tipus de problemes.	
	Temps				
	Funcional		La solució ha de ser implementada amb Oracle Coherence.	Resol el problema TSP (Traveling Salesman Problem) correctament.	
	Qualitat			Ha de ser una solució escalable i ràpida en donar resultats.	
	Mediambiental				

Figura 2.1: Taula de requisits

2.3 Recursos

Per a la realització d'aquest projecte es farà ús dels següents components:

- Software Oracle Coherence
- Dropbox
- SVN o quelcom per al control de versions
- Oracle Ibèrica CloudLab

En aquest treball es tindrà accés a un cloud privat (Oracle Ibèrica CloudLab) amb 2 nodes de còmput amb les següents característiques:

- Intel(R) Xeon(R) CPU E5620 @2.40Ghz (4 cores, 8 threads)
- 32GB de RAM
- 1 GB internal connectivity

Aquest hardware el facilita Oracle Ibèrica per poder fer les proves.

2.4 Planificació

2.4.1 Descomposició de processos en tasques

En aquest punt concretarem aquests processos en tasques suficientment controlables per fer la planificació temporal. Segons el cicle de vida on es troba la tasca:

1. Reunió amb el director per discutir sobre els objectius del projecte (Què es vol? Què farem?...)

2. Reunió amb el ponent per discutir i valorar els objectius del projecte

3. *Feedback* per part del professorat de GEP

4. Estudiar documentació de GEP (fer-ho més granulat seria menys controlable)

5. Abast del projecte

6. Planificació temporal

7. Pressupost

8. Presentació del document

9. Reunió amb el ponent per discutir sobre les diferents maneres cercades de resoldre el TSP.

10. Reunió amb el director per parlar sobre la reunió amb el ponent i buscar les millors opcions d'*Oracle Coherence* per resoldre el problema.

11. Cercar per diferents fonts (Internet, literatura, experts...) diferents maneres de resoldre TSP

12. Estudiar arquitectura de *Coherence*, possibilitats, mirar les possibles configuracions...

13. Pensar com modelar i dividir la metodologia trobada per tal de que sigui paral·lelitzable amb *Oracle Coherence*.

14. Implementar el problema TSP segons hem decidit prèviament.

15. Programació de l'entorn *Oracle Coherence*.

16. Integració de la modelització TSP amb l'entorn *Coherence*.

17. Reunió amb arquitecte d'Oracle i altres agents per veure la possibilitat d'adquirir dades reals.

18. Pensar com tractar, modificar, adaptar... les dades i/o la representació del TSP per poder provar-les en el nostre sistema.

19. Test de validació

20. Comprovar resultats, comentar-los, analitzar-los, descriure'ls...

Figura 2.2: Llistat de tasques

2.4.2 Gantt del projecte

		Nombre	Duración	Inicio	Fin	Predecesoras
1		Reunió amb el director per discutir sobre els objectius del	1d?	06/02/2013	06/02/2013	
2		Reunió amb el ponent per discutir i valorar els objectius de	1d?	12/02/2013	12/02/2013	1
3		Feedback per part del professorat de GEP	10d?	01/02/2013	14/02/2013	
4		Estudiar documentació de GEP (fer-ho més granulat seria	13d?	29/01/2013	14/02/2013	
5		Abast del projecte	1d?	30/01/2013	30/01/2013	
6		Planificació temporal	1d?	04/02/2013	04/02/2013	
7		Pressupost	1d?	06/02/2013	06/02/2013	
8		Presentació del document	1d?	15/02/2013	15/02/2013	2,3,4,5,6,7
9		Reunió amb el ponent per discutir sobre les diferents man	1d?	18/02/2013	18/02/2013	8,11
10		Reunió amb el director per parlar sobre la reunió amb el p	1d?	26/02/2013	26/02/2013	12,13
11		Cercar per diferents fonts (Internet, literatura, experts...) dif	20d?	21/01/2013	15/02/2013	
12		Estudiar arquitectura de Coherence, possibilitats, mirar les	20d?	29/01/2013	25/02/2013	
13		Pensar com modelar i dividir la metodologia trobada per ta	5d?	19/02/2013	25/02/2013	9
14		Implementar el problema TSP segons hem decidit prèvian	20d?	26/02/2013	25/03/2013	13
15		Programació de l'entorn Oracle Coherence.	7d?	27/02/2013	07/03/2013	10
16		Integració de la modelització TSP amb l'entorn Coherence.	20d?	26/03/2013	22/04/2013	14,15
17		Reunió amb arquitecte d'Oracle i altres agents per veure la	1d?	27/03/2013	27/03/2013	
18		Pensar com tractar, modificar, adaptar... les dades i/o la re	7d?	23/04/2013	01/05/2013	16,17
19		Test de validació	3d?	02/05/2013	06/05/2013	18
20		Comprovar resultats, comentar-los, analitzar-los, descriure	20d?	07/05/2013	03/06/2013	19

Figura 2.3: Definició de tasques

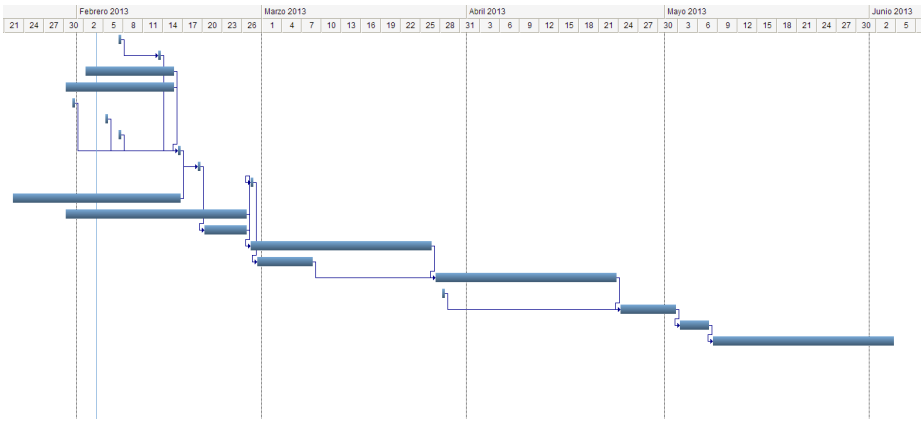

Figura 2.4: Gantt del projecte

2.4.3 Valoracions d'alternatives i pla d'accions

El projecte es planteja per ser finalitzat durant el segon quadrimestre 2012 - 2013. Per aquest motiu la planificació temporal està realitzada amb un marge

de temps suficientment gran per contemplar possibles problemes o desviacions del projecte. Com ja s'ha vist les dates de les tasques estan ajustades per ser acabades a principis de Juny deixant així aproximadament dues setmanes per si cap tasca necessités més temps.

Procés	Possibles obstacles	Protocol a seguir	Temps de marge
Generar documentació inicial	- Detectar problema o dubte en el disseny del projecte (inviabilitat, dificultat...) - *Feedback* negatiu respecte a la documentació	Comunicar-ho ràpidament al director/ponent o agents competents per tal de trobar una solució alternativa.	5 dies
Cerca de possibles mètodes de resolució pel TSP	Gran varietat de mètodes	Fitar les possibilitats a un grup reduït (els més famosos, els que tenen millor *performance*...)	1 dia
Preparació de l'entorn *Oracle Coherence*	- Documentació limitada (software especific i privatiu?) - Desconeixements dels millors paràmetres de configuració de l'entorn	Concretar una reunió amb experts *Oracle* en aquesta tecnologia per comentar els problemes trobats.	1 dia
Modelitzar el problema TSP per ser resolt amb *Oracle Coherence*	- Inviabilitat d'adaptar la metodologia escollida al *Grid Computing* - El problema creix molt en complexitat i/o hi ha *overhead* excessiu	En ambdós casos s'escollirà un altre mètode entre els possibles candidats.	2 dies

Figura 2.5: Variacions i alternatives 1

Implementació del TSP amb *Oracle Coherence*	- Documentació limitada o poc concisa pel nostre problema. Incapacitat per fer una correcta implementació - Resultats de *test* incorrectes	- Concretar una reunió amb experts *Oracle* en aquesta tecnologia per comentar els problemes trobats.	5 dies
Obtenció de dades per validació	- No es troben dades - Les dades no són adaptables al nostre problema	En aquest cas ens inventarem un escenari fictici però potencialment real	5 dies

Figura 2.6: Variacions i alternatives 2

Generació de dades per ser contrastades, valorades i explicades	- Els resultats no compleixen alguns dels requeriments - Impossibilitat de contrastar els resultats (amb quina solució suposadament òptima comparem?)	- Identificar la causa i trobar una solució. Ex: triga massa en donar els resultats --> Canviar metodologia Si no es troba cap alternativa es donaran els millors resultats obtinguts i es faran les conclusions adients. - Si no hi ha cap valor de referència per comparar resultats es donaran valors aproximats pel problema. Encara que no siguin reals ens proporcionarà una perspectiva de la qualitat de la solució.	15 dies

Figura 2.7: Variacions i alternatives 3

En el cas pitjor que totes les tasques tinguin problemes el temps estimat de marge sumaria al projecte 34 dies cosa que encara està dintre de l'objectiu establert.

2.5 Pressupost

A continuació es llisten les diferents tasques del projecte i els recursos utilitzats en cadascuna:

Tasca	Recursos vinculats	Costos directes	Cost total (x jornades de 8h)
1. Reunió amb el director per discutir sobre els objectius del projecte (Què es vol? Què farem?...)	→ 2 analistes	→2x40€/h = 80€/h	80 * 8 * 1 dia = 640€
2. Reunió amb el ponent per discutir i valorar els objectius del projecte	→ 1 hora com analista → 1 hora de personal professional	→2x40€/h = 80€/h	80 * 8 * 1 dia = 640€
3. Reunió amb el ponent per discutir sobre les diferents maneres cercades de resoldre el TSP	→ 1 hora com analista → 1 hora personal professional	→ 2x40€/h = 60€/h	60 * 8 * 1 dia = 480€
4. Reunió amb el director per parlar sobre la reunió amb el ponent i buscar les millors opcions d'*Oracle Coherence* per resoldre el problema	→ 2 analistes → Llicència *Oracle Coherence*	→ 2x40€/h = 60€/h → 5000€ x 2 (2 nodes) = 10.000€ (només la comptabilitzem un cop)	60 * 8 + 10.000 = 10480€

Figura 2.8: Pressupostos 1

5. Cercar per diferents fonts (Internet, literatura, experts...) diferents maneres de resoldre TSP	→ 1 analista	→ 40€/h	40 * 8 *20 = 6400€
6. Estudiar arquitectura de *Coherence*, possibilitats, mirar les possibles configuracions...	→ 1 analista → Llicència *Oracle Coherence*	→ 40€/h = 40€/h	40 * 8 *20 = 6400€
7. Pensar com modelar i dividir la metodologia trobada per tal de que sigui paral·lelitzable amb *Oracle Coherence*	→ 1 analista → Llicència *Oracle Coherence*	→ 40€/h	40 * 8 *5 = 1600€
8. Implementar el problema TSP segons hem decidit prèviament.	→ 1 programador	→ 20€/h	20 * 8 * 20 = 3200€
9. Programació de l'entorn *Oracle Coherence*.	→ Llicència *Oracle Coherence* → 1 programador	→ 20€/h	20 * 8 * 7 = 1120€

Figura 2.9: Pressupostos 2

10. Integració de la modelització TSP amb l'entorn *Coherence*	→ Llicència *Oracle Coherence* → 1 programador	→ 20€/h	20 * 8 * 20 = 3200€
11. Reunió amb arquitecte d'Oracle i altres agents per veure la possibilitat d'adquirir dades reals.			
12. Pensar com tractar, modificar, adaptar... les dades i/o la representació del TSP per poder provar-les en el nostre sistema.	→ 1 programador	→ 20€/h	20 * 8 * 7 = 1120€
13. Test de validació	→ Llicència *Oracle Coherence* → 1 programador	→ 20€/h	20 * 8 * 3 = 480€
14. Comprovar resultats, comentar-los, analitzar-los, descriure'ls...	→ Llicència *Oracle Coherence* → 1 programador	→ 20€/h	20 * 8 * 20 = 3200€

Figura 2.10: Pressupostos 3

Sumant les quantitats dona un cost directe del projecte de **38.960** euros.

2.5.1 Altres costos associat al projecte

Recurs	Preu
Portatil Dell	1.700€
Cloud Lab	4.000€

Figura 2.11: Altres costos del projecte

Total costos indirectes **5.700** euros.

Aquest pressupost s'ha realitzat amb costos aproximats.

2.6 Sostenibilitat i compromís social

Les solucions que es donen en aquest treball no tenen una solució negativa en l'àmbit social, ambiental i econòmic. Tot al contrari, la solució proposada té un impacte positiu en aquestes tres, al tractar-se d'optimitzar un problema per reduir costos del client. Per tant l'impacte econòmic positiu es evident. En `http://www.scienceofbetter.org/can_do/success_stories.htm` es poden trobar diferents casos exitosos on l'optimització a comportat un important benefici econòmic.

Pel que fa a l'àmbit social i ambiental, la forma en que afecta positivament a aquestes dues és similar. Al intentar reduir recursos (combustible en logística, material en electrònica, ...) es contribueix a minimitzar la subtracció de recursos naturals i per tan ajudem al medi ambient. L'àmbit social es recompensat ja que hi ha un impacte positiu econòmic i ambiental.

2.7 Lleis i regulacions

El present treball de fi de grau pot estar afectat per diferents lleis.

2.7.1 Documentació i bibliografia

Les 3 referències més consultades poden estar subjectes a diferents llicències:

- [1, 2] Aquestes dues referències estan subjectes a la llicència *copyright* del *MIT press*. No obstant aquests dos llibres han sigut comprats i pagats legalment.
- [3] Aquest article està a lliure disposició en la pàgina del autor `http://www.akira.ruc.dk/~keld/research/LKH/` i no està subjecte a cap llicència.

Apart d'aquest material, que ha sigut el més utilitzat, s'ha fet ús puntualment d'altres documents però sempre com a reforç i, en qualsevol cas, aquest treball no es basa en cap concepte o implementació que no surti en els esmentats anteriorment.

2.7.2 Software

El software també pot estar lligat a diferents lleis que poden afectat aquest treball:

- Eclipse SDK, aquest IDE està sota la llicència Eclipse Public License Version 1.0 [2] i pot ser descarregat de forma gratuïta en la seva pàgina oficial [3].
- Oracle Coherence, està disponible sota la llicència OTN[4] (Oracle Technology Network Developer License Terms) i ens limitem a l'entrada que diu: "We grant you a nonexclusive, nontransferable limited license to use the programs only for the purpose of developing, testing, prototyping and demonstrating your application, and not for any other purpose".

[2] `http://www.eclipse.org/legal/epl-v10.html`
[3] `http://www.eclipse.org/downloads/`
[4] `http://www.oracle.com/technetwork/licenses/standard-license-152015.html`

Molts dels problemes amb els quals ens trobem i necessitem resoldre són NP-complete. Aquests problemes però, tenen molta utilitat a la pràctica i no ens podem permetre el luxe de no trobar solucions pel simple fet de no poder-ho fer en temps polinòmic. Hi ha, com a mínim, tres maneres diferents de intentar resoldre aquest tipus de problemes. La primera, si l'entrada del problema es tan petita que inclús amb l'algorisme exponencial podem trobar les solucions òptimes en un temps raonable. Segon, podem aïllar instàncies del problema les quals podem resoldre en temps polinòmic. Tercer, podem fer aproximacions per intentar buscar solucions *aproximades* en temps polinòmic. Aquest tipus d'aproximacions són, sovint, suficientment bones com per estudiar-les. Anomenarem a aquest tipus d'algorismes *algorismes aproximats*.

La descripció del desenvolupament del treball s'estructura de la següent manera. Al capítol 3 descriurem les implementacions seqüencials fetes dels 3 algorismes seleccionats, els experiments fets, i la comparativa. Al Capítol 4 descriurem les implementacions paral.leles de AOC i LKH, els experiments i fets i, de nou, la comparativa. Al Capítol 5 presentarem les conclusions de la feina realitzada.

Part II

Cos del treball

Capítol 3

Solucions seqüencials

3.1 Resolució mitjançant un arbre d'expansió mínim

Un arbre d'expansió mínim d'un graf $G = (V, E)$, on V és el conjunt de nodes, E el conjunt de possibles conexions entre aquests i per cada aresta $(u, v) \in E$ tenim un cost associat $w(u, v)$, és un subconjunt $T \subseteq E$ que connecta tots els nodes i que el cost

$$w(T) = \sum_{(u,v)\in T} w(u, v) \tag{3.1}$$

és mínim. [1]

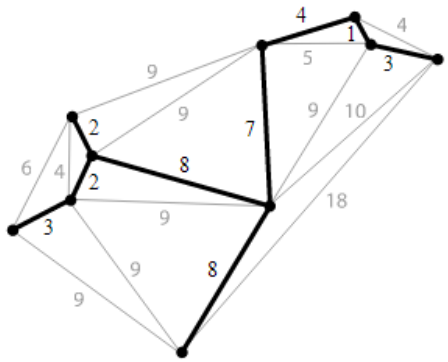

Figura 3.1: Exemple d'arbre d'expansió mínim

`http://en.wikipedia.org/wiki/Minimum_spanning_tree`

[1] Per més informació sobre arbres d'expansió mínim poder anar a [1]

3.1.1 Rati dels algorismes d'aproximació

Dintre dels algorismes d'aproximació hi ha uns que s'aproximen un cert rati a l'òptim. Direm que aquests algorismes tenen un rati d'aproximació $p(n)$ [2] si, per qualsevol entrada de grandària n, el cost C de la solució obtinguda per l'algorisme està dintre del factor de $p(n)$ de C^* d'una solució òptima:

$$max(C/C^*, C^*/C) \leq p(n) \tag{3.2}$$

Aquesta definició d'aproximació serveix tan per problemes de minimització com maximització. Per a problemes de maximització, $0 < C \leq C^*$, i el rati C^*/C ens dona el factor d'aproximació el qual la solució òptima és més gran que el cost aproximat de la solució. Similarment, per a problemes de minimització, $0 < C^* \leq C$, i el rati C/C^* ens dona el factor el qual el cost de la solució aproximada és més gran que l'òptim, ja que assumim que els costos són positius la formula anterior està sempre ben definida.

Per a molts problemes hi ha algorismes d'aproximació d'aquest tipus amb un rati petit i constant. Per altres problemes però, aquest rati creix amb la mida de l'entrada [3].

Hi ha altres problemes on podem negociar entre temps d'execució i el rati d'aproximació [4]. Un *esquema d'aproximació* per un problema d'optimització és un algorisme on la seva entrada no és només l'entrada del problema si no que també li passem un valor $\epsilon > 0$ tal que per cada ϵ, l'esquema és un algorisme $(1+\epsilon)$-aproximat. Si aquest esquema s'executa en temps polinòmic fixada una ϵ en direm que és un *esquema d'aproximació en temps polinòmic*.

3.1.2 Arbre d'expansió mínim com a resolució del TSP

Podem modelar el TSP com un problema de resoldre un arbre d'expansió mínim. Més endavant mostrarem que aquesta aproximació és una solució *2-aproximada*.

Primer de tot trobarem un arbre d'expansió mínim sobre un graf. El cost d'aquest arbre ens donarà una fita inferior del cost del tour òptim pel TSP. Utilitzarem aquest arbre per crear un tour tal que el seu cos no serà dues vegades el cost de l'arbre trobat. A continuació hi ha, en pseudocodi, un algorisme d'alt nivell que fa el que volem:

[2] Podem dir simplement p en el cas que el factor no depengui de n
[3] Com és el cas del problema *set-cover*
[4] Per exemple en el problema *subset-sum*

Algorithm 1 Aproximació del TSP mitjançant la resolució d'un arbre d'expansió mínim

Require: Graf $G = (V, E)$
Ensure: Cicle hamiltonià o tour H

1: selecciona un vèrtex $x \in G.V$ com a arrel
2: troba un arbre d'expansió mínim T de G des de l'arrel x
3: Sigui H una llista de vèrtex, ordenats segons preordre de T
4: **return** el cicle hamiltonià H

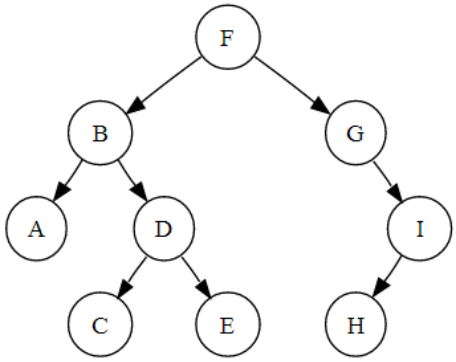

Figura 3.2: Exemple d'arbre

`http://en.wikipedia.org/wiki/Tree_traversal`

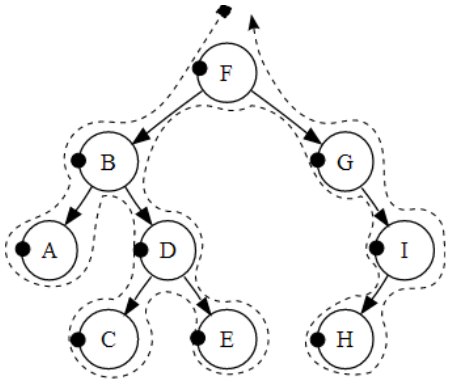

Figura 3.3: Exemple d'un camí en preordre

`http://en.wikipedia.org/wiki/Tree_traversal`

El cost algorítmic d'aquest codi depèn de la implementació per trobar l'arbre d'expansió mínim. Si ho fem mitjançant l'algorisme prim el cost és fitat per $O(n^2)$, Kruskal per $O(El\log E)$, etc. En definitiva podem resoldre aquest problema en temps polinòmic. A continuació mostrarem que aquest algorisme ens dona solucions que mai seran pitjors que dues vegades el cost del tour òptim.

Sigui H^* el tour òptim d'un graf qualsevol. Obtenim un arbre d'expansió de cost mínim associat a aquest graf mitjançant l'extracció d'una aresta qualsevol (mirar Figura 3.2 per aquest exemple). Donat això, l'arbre T que retorna l'algorisme mostrat anteriorment ens dona una fita inferior del valor del cost del tour òptim:

$$c(T) \leq c(H^*) \tag{3.3}$$

Un recorregut sencer de T llista els vèrtexs quan han sigut visitats i també quan acaben de recórrer qualsevol subarbre. Anomenem aquest recorregut W. El recorregut sencer del nostre exemple mostrat a la Figura 3.3 seria:

$$F, B, A, B, D, C, D, E, D, B, F, G, I, H, I, G, F$$

Aquest camí passa per cada node dues vegades, es a dir, cada aresta serà travessada dues vegades en el nostre recorregut. Així doncs podem afegir a la nostre definició la següent equació:

$$c(W) = 2c(T) \tag{3.4}$$

amb la qual cosa podem també deduir:

$$c(W) \leq 2c(H^*) \tag{3.5}$$

i per tant, el cost del recorregut W serà com a màxim dues vegades el cost òptim.

No obstant, el recorregut W rarament serà un tour ja que segurament visitarà més d'un cop cada node. Gràcies a que només ens fixarem en problemes mètrics (que compleixen la inequació triangular), podem esborrar els vèrtexs repetits i quedar-nos només amb la primera aparició de cadascun amb la seguretat que no augmentarem el cost associat. D'aquesta manera el tour que obtindrem del exemple anterior serà:

$$F, B, A, D, C, E, G, I, H$$

Si H és el recorregut retornat pel nostre algorisme, podem afegir l'equació:

$$c(H) \leq c(W) \tag{3.6}$$

On, junt amb les altres, obtenim:

$$c(H) \leq 2c(H^*) \tag{3.7}$$

Amb la qual cosa demostra que les solucions retornades pel nostre algorisme mai seran pitjor que dues vegades el tour òptim.

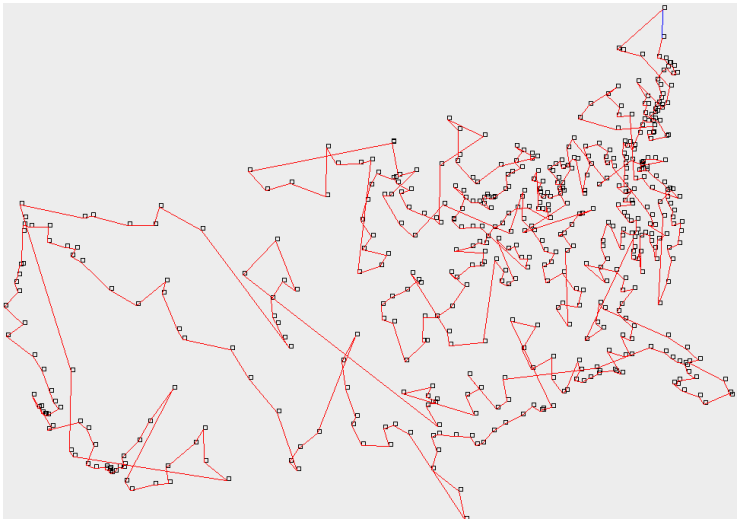

Figura 3.4: Mapa EUA obtingut mitjançant l'MST- att532

3.2 Resolució mitjançant l'algorisme Lin-Kernighan Helsgaun

3.2.1 Descripció

L'heurístic Lin-Kernighan [15] està considerat com l'algorisme més efectiu per resoldre instàncies del TSP. És un algorisme que, a diferència de l'Ant colony i altres heurístics, està desenvolupat per resoldre només instàncies del TSP simètric. En anys posteriors s'han proposat millores sobre l'algorisme de Lin-Kernighan original. En aquest treball estudiarem la millora que va fer Keld Helsgaun [3] a aquest algorisme.

En el primer punt explicarem l'algorisme Lin-Kernighan en la seva versió original. Seguidament comentarem les aportacions que va fer Helsgaun per millorar l'eficiència del original i per últim mirarem els resultats que ens aporta aquest algorisme. Per la implementació no s'ha llegit l'article original sinó les descripcions que es troben a [3], més accessibles.

3.2.2 Lin-Kernighan

Aquest algorisme està classificat en els heurístics de millora d'un tour. Es a dir, a partir d'un tour inicial aplica canvis per obtenir nous tours que siguin millors. Lin-Kernighan pertany a la família d'algorismes *2-opt*. Aquests tipus d'algorismes són un cas concret d'un més general anomenat algorismes λ-*opt* on la λ és un enter positiu més gran que 0. El que fan aquests tipus d'algorisme és intentar treure λ arestes del tour inicial i afegir λ noves arestes per aconseguir un tour millor. El concepte de λ-*opt* esdevé de λ-*optimality*: [3]

Definition 1. *Un tour és λ-òptim o λ-opt si és impossible d'obtenir millors tours reemplaçant λ arestes per λ noves arestes.*

Es conclou de la definició anterior que si un tour és λ-opt també ho serà per λ' tal que $1 <= \lambda' <= \lambda$

No obstant, les operacions que s'han de fer per provar tots els valors de λ creix ràpid amb la grandària de l'entrada, aproximadament $O(n^{\lambda})$ [3] en implementacions senzilles de fer. Per aquest motiu la majoria d'aquests algorismes utilitzen una λ de l'ordre de 2, 3, 4 o 5 amb restriccions de cerca més grans conforme augmenta λ. Lin-Kernighan però, determina el valor d'aquesta λ en temps d'execució i no té una fixada des de un principi.

Així doncs Lin-Kernighan intenta aplicar repetidament canvis a un tour inicial per tal de reduir el cost associat a aquest fins que una certa condició es compleix o ja no es pot millorar més.

Formalment: sigui T un tour, l'algorisme intenta cercar dos conjunts d'arestes $X = x_i, ..., x_r$ i $Y = y_1, ..., y_r$ tal que si traiem les arestes de X i posem les de Y, el tour resultant és millor. Aquest canvi d'arestes s'anomena moviment *r-opt*.

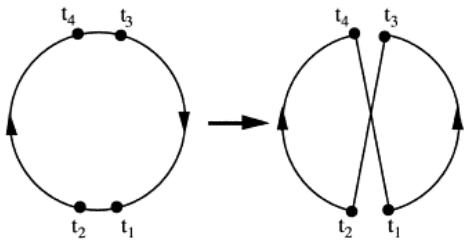

Figura 3.5: Moviment 2-opt

`http://www.sciencedirect.com/science/article/pii/S0377221799002842`

Aquests dos conjunts són construïts element per element. A cada iteració i s'afegeix un nou element x_i i y_i en X i Y respectivament.

Per tal de millorar l'eficiència de l'algorisme es requereix que aquests canvis compleixin unes certes propietats:

- El criteri de seqüencialitat. x_i i y_i han de tindre un node en comú i també y_i i x_{i+1}. Si t_1 és un dels nodes de x_1 tenim que $x_i = (t_{2i-1}, t_{2i}$, $y_i = (t_{2i}, t_{2i+1})$ i $x_{i+1} = (t_{2i+1}, t_{2i+2})$ per tot $i >= 1$. Es a dir, els nous possibles canvis que es poden afegir als conjunts queden limitats pels canvis anteriors. D'aquesta manera acotem molt la cerca i augmentem l'eficiència.

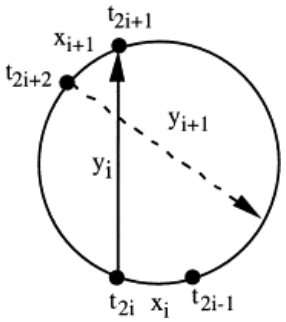

Figura 3.6: El moviment y_{i+1} ha de tenir un dels nodes extrem de x_{i+1}: t_{2i+2} o t_{2i+1}

`http://www.sciencedirect.com/science/article/pii/S0377221799002842`

La cadena de canvis serà del estil:

$$(x_1, y_1, x_2, y_2, x_3, ..., x_r, y_r) \equiv ((t_1, t_2), (t_2, t_3), (t_3, t_4), (t_4, t_5), ..., (t_{2r-1}, t_{2r}), (t_{2r}, t_1))$$

Una condició necessària però no suficient és que aquesta cadena de canvis sigui seqüencial. Es a dir, l'últim element de Y és del estil (t_{2r}, t_1)

Hi ha a vegades però que no sempre es pot obtenir el tour òptim mitjançant canvis seqüencials. La imatge que hi ha a continuació és un exemple d'un canvi que no es pot fer de forma seqüencial:

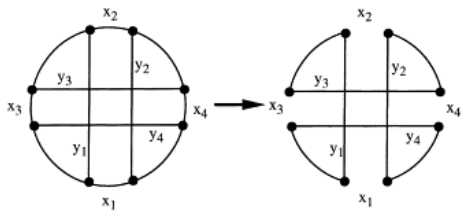

Figura 3.7: Canvi no seqüencial

http://www.sciencedirect.com/science/article/pii/S0377221799002842

- El criteri de factibilitat. Donat un canvi $X_i = (t_{2i-1}, t_{2i})$ si ajuntem t_{2i} amb t_1 dona com a resultat un tour. Només en el cas que $i = 2$ deixarem que un canvi no compleixi aquesta propietat.
- El criteri del guany positiu. y_i ha de ser escollit tal que el guany total G_i sigui positiu. Es a dir, si $g_i = c(x_i) - c(y_i)$ és el guany que s'obté al canviar l'aresta x_i per y_i llavors $G_i = g_1 + g_2 + ... + g_i$ ha de ser positiu. Aquesta condició pot ser molt restrictiva però no ho és tan ja que qualsevol seqüència de sumes te una permutació dels seus sumands tal que cada suma parcial és positiva [10].
- El criteri de disjuntivitat. Els conjunts X i Y han de ser disjunts.

Per comprovar que un canvi satisfà la propietat de seqüencialitat podem fer-ho només amb la informació de les arestes involucrades en un canvi. Es a dir, si $X = (x_1, x_2, x_3)$ i $Y = (y_1, y_2, y_3)$ podem comprovar si el resultat d'aplicar aquest 3-mov és un tour o no seguint aquesta casuística:

Sigui t_r el vèrtex de partida per construir x_i hi ha dos opcions per fer-ho ja que el criteri de seqüencialitat ens exigeix que t_{r+1} sigui predecessor o successor de t_r però només un d'ells complirà el criteri de factibilitat. No obstant, pel cas de $i = 2$, els dos canvis es poden fer:

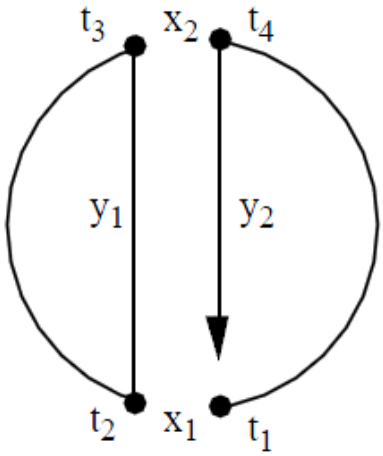

Figura 3.8: y_2 que no compleix criteria de satisfactibilitat

http://www.sciencedirect.com/science/article/pii/S0377221799002842

Si t_4 hagués caigut entre t_2 i t_3 el resultat hauria sigut un tour. En canvi si després del moviment mostrat a la imatge t_5 cau entre t_2 i t_3 hi ha dues opcions per escollir x_3.

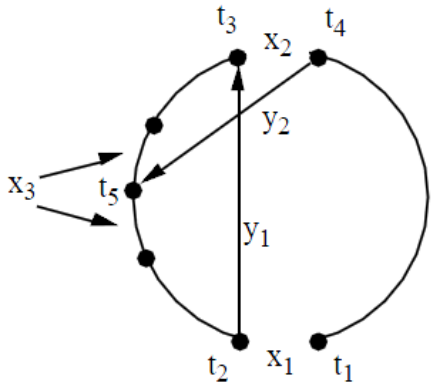

Figura 3.9: Dos vèrtex possibles per t_6

http://www.sciencedirect.com/science/article/pii/S0377221799002842

Si t_5 però, recau entre t_1 i t_4, només hi ha una única opció per t_6 i t_7:

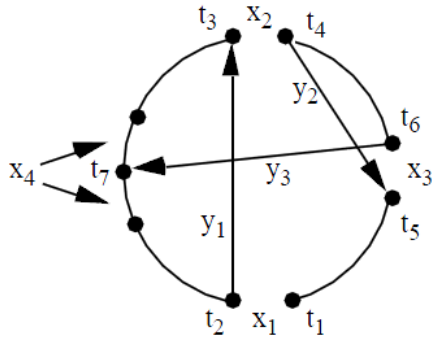

Figura 3.10: Única opció t_6 i t_7 i dos per t_8

`http://www.sciencedirect.com/science/article/pii/S0377221799002842`

En els annèxos hi ha un exemple per comprovar si un canvi genera un tour només amb la informació dels nodes afectats 6.1.4

Per limitar encara més la cerca dels dos conjunts X i Y l'algorisme original utilitza les següents regles:

- La cerca d'un link per ser inserit, $y_i = (t_{2i}, t_{2i+1})$, es limita als 5 veïns més propers a t_{2i}.
- Per $i >= 4$, no es poden afegir links a X que pertanyin als (2-5) millors tours
- Si el tour resultant d'una cerca és el mateix tour que l'anterior l'algorisme para.

La primera regla addicional i la segona són heurístics que acoten la cerca però poden fer que no es sigui capaç de trobar l'òptim. Helsgaun però, fa millores en aquest sentit que després veurem.

A part d'aquestes regles que s'apliquen per acotar la cerca hi ha altres que ajusten la direcció de la cerca:

- Per un possible link $y_i (i >= 2)$, s'escull el que millor guany $c(x_{i+1}) - c(y_i)$ ens doni.
- Si hi ha dos alternatives per x_4, agafem el que més cost $c(x_4)$ tingui

Per últim quan l'algorisme troba un òptim local l'intenta millorar a partir de *4-opt* no seqüencials.

3.2.3 Lin-Kernighan Helsgaun

Les modificacions que va afegir Helsgaun varen ser bàsicament a les regles heurístiques que s'utilitzaven per acotar i ajustar la direcció de la cerca. A continuació s'exposen aquestes modificacions.

Candidate sets

Una de les regles més important de l'algorisme era la limitació que es feia als 5 veïns més propers. Aquesta regla segurament dona com a resultat tours curts però és possible que no puguem arribar al òptim. Per exemple, en un *dataset* de 532 ciutats una de les arestes del tour òptim utilitza el veí número 22 [3]. Per tan aquesta regla ens limita des d'un principi el trobar o no tours òptims.

Helsgaun introdueix un concepte de mesura α-*nearness* que indica la bondat d'una aresta de pertànyer al tour òptim. Per aquesta mesura utilitzarem un concepte que es defineix a continuació:

Definition 1. *Un* 1-tree *d'un graf* $G = (N, E)$ *és un arbre d'expansió del conjunt* $N \setminus 1$ *combinat amb dues arestes de* E *incidents al node* 1.

El node que s'exclou per l'arbre d'expansió mínim és arbitrari i no te perquè ser l'1. S'observa que un 1-tree no és un arbre ja que conte un cicle:

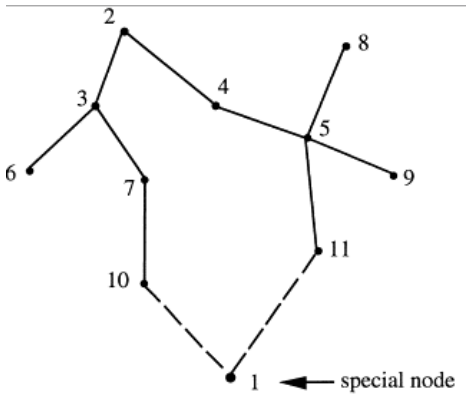

Figura 3.11: 1-tree

http://www.sciencedirect.com/science/article/pii/S0377221799002842

Podem doncs observar que:

- Un tour òptim és un mínim 1-tree on el grau de tots els nodes és 2.
- Si un mínim 1-tree és un tour, llavors el tour és òptim.

Per tan podem formular el problema del TSP simètric com a: *Cercar un 1-tree mínim tal que tots els nodes tinguin grau 2*

Generalment un arbre d'expansió mínim conté moltes arestes en comú amb un tour òptim. Un tour òptim normalment conté el 70 i 80 per cent de les arestes d'un 1-tree [3]. Per tant, els 1-tree semblen una bona mesura heurística per determinar si una aresta ha de pertànyer o no a un tour òptim. Els costos de les arestes doncs queden modificats amb aquest nou concepte de α-*nearness*.

Més formalment, aquesta mesura es pot definir com:

Definition 1. *Sigui T un 1-tree mínim de cost associat $L(T)$ i sigui $T^+(i,j)$ el 1-tree mínim que conté l'aresta (i,j). Llavors l'*α-nearness *d'una aresta (i,j) es defineix com la quantitat:*

$$\alpha(i,j) = L(T^+(i,j)) - L(T). \tag{3.8}$$

Per tan podem agafar les k α-*nearest* arestes per determinar el conjunt de candidats de cada node.

De la definició anterior per calcular el valor α-*nearness* podem establir les següents regles per calcular $T^+(i,j)$:

- Si (i,j) pertany a T, llavors $T^+(i,j)$ és igual a T
- Si (i,j) te un 1 en un dels seus nodes ($i = 1 o j = 1$), llavors $T^+(i,j)$ s'obté de T reemplaçant l'aresta incident a 1 amb cost associat més gran per (i,j)
- Si no, s'insereix (i,j) a T. Això crea un cicle que conté (i,j) en l'arbre d'expansió mínim. $T^+(i,j)$ s'obté traient l'aresta amb cost associat més gran d'aquest cicle.

Els dos primers casos són fàcils de calcular i es pot arribar a fer en temps constant. En el tercer cas podem utilitzar una relació dels diferents valors α:

Sigui $\beta(i,j)$ el cost de l'aresta a treure del cicle que es forma quan s'introdueix (i,j), llavors $\alpha(i,j) = c(i,j) - \beta(i,j)$.

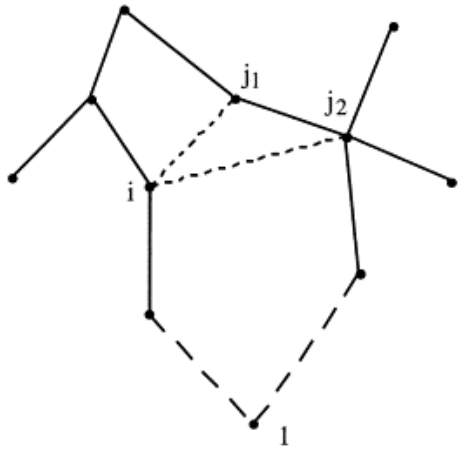

Figura 3.12: $\beta(i,j_2)$ es pot obtenir mitjançant $\beta(i,j_1)$

`http://www.sciencedirect.com/science/article/pii/S0377221799002842`

Si (j_1,j_2) és una aresta de l'arbre d'expansió mínim de la figura anterior, $\beta(i,j_2)$ pot ser calculada com el màxim de $\beta(i,j_1)$ i $c(i_1,j_2)$. 6.1.1

Els valors α semblen ser un bon estimador per indicar la pertinença o no d'una aresta al tour òptim. A α's més petites, més probabilitat hi ha de que aquella aresta estigui en el tour òptim. Per exemple, pel problema de les 532 ciutats necessitàvem, en el cas pitjor, arribar fins el veí 22 d'un node per aconseguir

l'òptim, mentre que si utilitzem els valors α la posició baixa fins el número 14 [3].

Encara que aquesta nova aproximació sigui millor que l'original, els valors α encara poden ser millorats si fem un canvi a la matriu de costos del graf. La transformació es basa en les següents observacions [3]:

- Cada tour és un 1-tree. Per tant el cost associat al 1-tree mínim és un límit inferior del cost del tour òptim.
- Si tots els costos de les arestes són augmentats per un valor π_i depenen del node i incident, qualsevol solució òptima d'aquesta modificació també ho serà de l'original. Per tant si els costos de la matriu $C = (c_{ij})$ és transformat a un altre $D = (d_{ij})$, on

$$d_{ij} = c_{ij} + \pi_i + \pi_j \tag{3.9}$$

  llavors un tour òptim de D també ho serà per C.
- Si T_π és un 1-tree de cost associat mínim de D, llavors el seu cost $L(T_\pi)$, és un límit inferior del tour òptim de D i, conseqüentment, $w(\pi) = L(T_\pi) - 2\sum \pi_i$ és un límit inferior del cost del tour òptim de C.

L'objectiu ara és trobar una transformació, $C \mapsto D$, tal que el vector $\pi = (\pi_1, \pi_2, ..., \pi_n)$ maximitza el límit inferior $w(\pi) = L(T_\pi) - 2\sum \pi_i$.

La maximització de $w(\pi)$ no és fàcil de trobar ja que és una funció *piecewise* lineal [5] i còncava. Per aquest motiu s'utilitza una optimització per *subgradient* [11]. Es tracta de dirigir el vector π en direcció del subgradient, es a dir, $\pi^{k+1} = \pi^k + t^k v^k$ on v^k és el subgradient del vector i t^k un escalar positiu anomenat *step size*.

Per el nostre problema de maximització el subgradient be donat pel grau dels nodes $v^k = d^k - 2$ [3], on d^k és un vector que conté els graus dels nodes del 1-tree en el pas k. Es pot veure que la resta del -2 en el vector del subgradient fa que el algorisme vagi cap a 1-tree amb nodes de grau 2. Està demostrat que aquest procés sempre convergeix [11].

Aquesta modificació dels costos de la matriu original millora el valor α calculat. En el problema de les 532 ciutats, per exemple, totes les arestes de l'òptim estan entre els 5 α veïns més propers [3]. En l'annex hi ha la funció principal implementada 6.1.2.

[5] Una funció piece-wise és una funció formada per línies rectes consecutives de diferent pendent

Sel·lecció de les arestes a treure

Els valors α són útils per acotar el nombre d'elements a examinar per sel·leccionar el conjunt Y. Similarment, Helsgaun també incorpora un acotament dels elements del conjunt X:

- El primer element d'X, $x_1 = (t_1, t_2)$ no ha de pertànyer al millor tour trobat. Al principi, quan cap tour ha sigut trobat, x_1 no ha de pertànyer al 1-tree mínim calculat.
- L'últim element a afegir a X no ha de estar en el conjunt de canvis a fer. Aquesta regla serveix per evitar bucles de canvis en l'algorisme.

Moviments Bàsics

En la nova versió de l'algorisme el moviment bàsic és el 5-opt seqüencial. Es a dir, l'algorisme intentarà millorar el tour aplicant canvis de 5 en 5. No obstant, si en el camí de trobar un 5-opt, troba un 2-opt, 3-opt o 4-opt que millora el tour, l'algorisme para i retorna el nou tour trobat.

Al contrari que l'algorisme original que intentava fer canvis 2-opt o 3-opt (amb excepcions d'uns casos que intentava aplicar 4-opt) el nou algorisme es basa en canvis 5-opt. Podria semblar que això és un impacte molt gran al rendiment però, donat que acotem la cerca amb els 5 veïns més propers, la pèrdua de rendiment és molt més petita.

Una altre variació en la cerca dels elements de X i Y és que els moviments no seqüencials no són únicament canvis 4-opt si no que s'apliquen les següents regles:

- Un canvi 2-opt que no genera una solució factible, seguida d'un moviment 2-opt o 3-opt que generi una solució factible, és un moviment no seqüencial vàlid.
- Un canvi 3-opt que no genera una solució factible, seguida d'un moviment 2-opt el qual genera una solució factible, és un moviment no seqüencial vàlid.

Aquets canvis no seqüencial es fan de la mateixa manera que els seqüencials. A diferencia de l'algorisme original que només ho feia com a post-optimització.

Tour inicial

Un tema important és la selecció del tour inicial amb el qual aplicar Lin-Kernighan. L'algorisme original genera un tour aleatori. La literatura actual sobre el tema ens diu que trobar o no l'òptim no depèn massa del tour inicial però podem arribar a estalviar temps amb tours que són més propers a l'òptim.

En la nova implementació de Lin-Kernighan, Helsgaun construeix el tour inicial de la següent manera:

Algorithm 2 Pseudo-codi per la generació del tour inicial

Escull un node inicial i aleatori
2: Selecciona un node j, que no ha sigut escollit abans, segons:
(a) (i, j) és una aresta candidata // i te entre els 5 veïns a j
4: (b) $\alpha(i, j) = 0$
(c) (i, j) pertany al millour tour trobat en aquell moment.
6: Si no es possible, escull una j tal que (i,j) està en les arestes candidates
Si tampoc es possible, selecciona una j que no ha sigut prèviament escollida.

8: $i = j$. Si encara falten nodes per escollir anar al pas 2.

En els annexos hi ha la implementació en java que s'ha fet per la generació del tour inicial. En l'Annex 6.1.3 es troba la implementació.

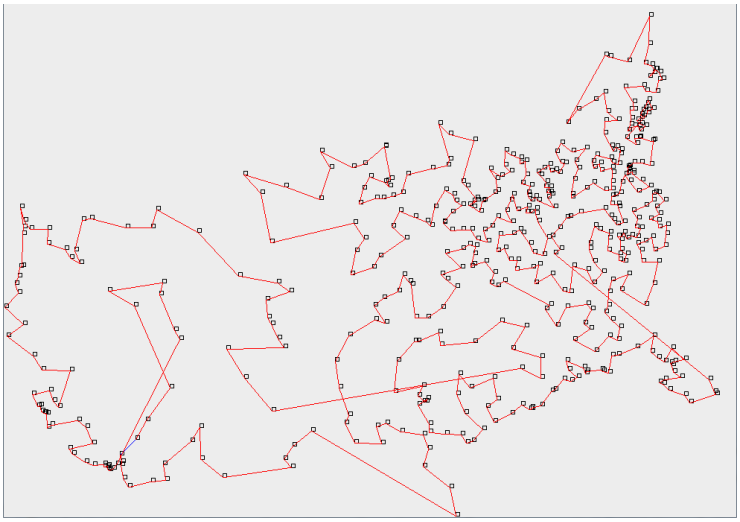

Figura 3.13: Mapa EUA obtingut mitjançant Lin-Kernighan Helsgaun - att532

3.3 Resolució mitjançant l'algorisme Ant Colony

3.3.1 Descripció

L'algorisme ACO (Ant Colony) s'inspira en el comportament de les formigues biològiques per trobat un camí mínim entre el formiguer i el menjar.

Les colònies de formigues, i en general els insectes que viuen en societat, són agents independents que, encara que siguin simples individualment, poden fer tasques relativament complexes si cooperen entre ells per tal de superar la capacitat individual de cadascuna.

La idea principal dels algorismes que intenten simular aquest tipus de comportament és imitar els principis de coordinació que segueixen aquests agents per tal de treballar per un fi en comú de forma eficient. Aquests principis es basen en diferents aspectes però un dels més importants és la comunicació. La majoria d'aquests insectes socials utilitzen l'*stigmergy*, una forma indirecta de comunicació mitjançant modificacions de l'entorn. Per exemple, hi ha formigues que depositen substàncies químiques anomenades feromona a terra de tal manera que augmenten la probabilitat que una altre formiga passi per aquest mateix camí, creant així un camí de feromones.

L'objectiu doncs és modelar el comportament de les formigues per aplicar-ho en la resolució del TSP. Una primera aproximació es pot fer amb l'experiment del doble pont.

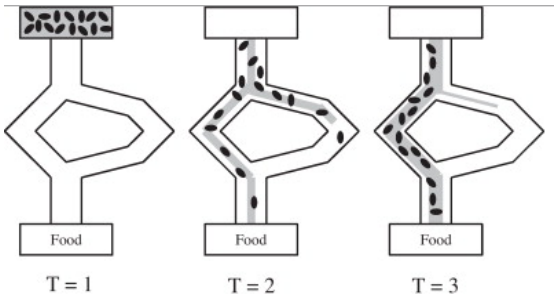

Figura 3.14: Experiment del doble pont

`http://www.sciencedirect.com/science/article/pii/S1568494613000264`

Al començament no hi ha cap feromona en els dos possibles camins. La probabilitat que les primeres formigues escollin un camí o l'altre és la mateixa. Conforme les formigues intenten buscar el menjar les que arriben abans (les del camí més curt) tornen pel mateix camí deixant en el seu pas feromones. Així doncs a partir d'un cert temps la probabilitat de que una formiga agafi el camí curt serà més gran que l'altre. D'aquesta manera les formigues acaben convergint en el camí més curt.

Les feromones que deixen les formigues biològiques sofreixen d'una evaporació en el temps. Aquesta evaporació però, no te molt impacte. En canvi s'ha vist

en els experiments que les formigues artificials requereixen d'aquest procés per evitar que l'algorisme convergeixi ràpidament a solucions sub-òptimes [2].

Amb una mica més de formalitat, sigui Ψ el número de formigues que creuen el pont a una velocitat constant de v cm/s, deixant una unitat de feromona en el camí. Sigui l_s i l_l el camí curt (short) i llarg (long) respectivament, una formiga que escull el camí curt travessarà el pont en $t_s = l_s/v$ segons, mentre que una formiga que escull el camí llarg o farà en $r * t_s$ segons, on $r = l_l/l_s$. Així doncs la probabilitat $p_s(t)$ de escollir en la primera bifurcació el camí curt és:

$$p_s(t) = \frac{(t_s + \varphi_s(t))^\alpha}{(t_s + \varphi_s(t))^\alpha + (t_s + \varphi_l(t))^\alpha} \tag{3.10}$$

on $\varphi_l(t)$ és el número de feromones a l'instant t, α és un paràmetre on diferents experiments han comprovat que per un $\alpha = 2$ es troben millors resultats i $p_t(t)$ es pot calcular a partir de la igualtat:

$$p_s(t) + p_l(t) = 1 \tag{3.11}$$

Hi ha moltes variants i implementacions de l'algorisme ant colony. La exposada en aquest treball s'aproxima a l'Ant System.

3.3.2 Modelització del ACO per al TSP

En l'algorisme ACO pel TSP les feromones s'associen als arcs, es a dir, π_{ij} indica la prioritat de visitar j després de visitar i. L'heurístic es defineix com a $\eta_{ij} = 1/d_{ij}$ i per tant la probabilitat de visitar i després de j es inversament proporcional a la distancia entre les dues. Si hi ha un cas que $d_{ij} = 0$ la informació de l'heurístic η_{ij} s'assigna a un valor molt petit.

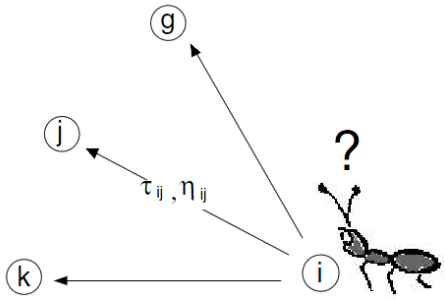

Figura 3.15: Formiga en la ciutat i que ha de escollir entre k, j i g amb la informació de l'heurístic i les feromones

Els tours són construïts aplicant el següent procediment:

- S'escull una ciutat de partida de la qual sortirà la formiga
- S'utilitza la informació de l'heurístic i el camí de feromones per, progressivament, afegir nodes que la formiga no ha visitat fins que totes les ciutats han sigut visitades.
- La formiga torna a la ciutat inicial.

Després que les formigues han tornat al punt de partida, depositen feromones en el camí que han seguit.

Construcció del tour

En la nostre implementació les formigues construeixen concurrentment el tour. Inicialment les formigues comencen en una ciutat aleatòria. A cada pas, la formiga k aplica una regla probabilística, anomenada regla proporcional aleatòria, per decidir quina ciutat visita després. En particular, la probabilitat amb la qual la formiga k, en la ciutat i, escull anar a la ciutat j és [6]:

$$p_{ij}^k = \begin{cases} \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in N_i^k} [\tau_{il}]^\alpha [\eta_{il}]^\beta} & \text{si } j \in N_i^k \\ 0 & \text{si } j \notin N_i^k \end{cases} \tag{3.12}$$

on $[\eta_{ij}] = 1/d_{ij}$ és la informació heurística que tenim a priori, α i β són uns paràmetres per donar més pes a la informació heurística o a les feromones, i N_i^k és el conjunt de veïns disponibles en la ciutat i per la formiga k, es a dir, el conjunt de ciutats que encara no ha visitat. El paper que juga els paràmetres α i β és el següent: Si $\alpha = 0$ les ciutats que són més properes (la seva distància és menor) són les que seran escollides. En aquest sentit l'algorisme es comportaria com un algorisme voraç. Si $\beta = 0$, només s'utilitza la informació de les pheromones sense cap informació heurística. Això dona sovint a resultats dolents i conforme $\alpha > 1$ es dona més ràpidament la situació d'*stagnation*, es a dir, una situació on totes les formigues segueixen el mateix camí i construeixen el mateix tour [2].

En la implementació feta, per tal de escollir una ciutat en concret donades les probabilitats trobades de la forma anterior, s'ha fet de la següent manera 6.2.1:

Sigui $n = 4$ el número de ciutats. Una vegada les probabilitats han sigut calculades, tindrem un array de mida n tal que:

0.15	0.10	0.35	0.40

[6] Podeu veure la seva implementació en l'Annex 6.2.1

ara agafem aquestes probabilitats i fem que en cada posició de l'array estigui la probabilitat acumulada:

0.15	0.25	0.60	1

seleccionem un número aleatori $0 <= r <= 1$ i recorrem l'array. Sigui p_i la probabilitat acumulada en la posició i, quan $r >= p_i$ i $r <= p_{i+1}$ llavors seleccionem aquesta ciutat com la candidata a ser visitada.

Aquest mètode s'anomena *roulette wheel selection* i és similar a una ruleta ja que cada ciutat ocupa tantes caselles d'aquesta com la seva probabilitat entre la probabilitat total (ha sigut normalitzada). En el nostre cas hi ha tantes caselles com ciutats però amb la condició per seleccionar una ciutat agafem un interval que és més gran o més petit segons la probabilitat de la ciutat.

Actualització del camí de feromones

Quan cada formiga ha construït el seu tour, el camí de feromones s'actualitza. Primer de tot decrementem el valor de totes les arestes un cert factor (el factor d'evaporació de les feromones) i després afegim una certa quantitat de feromones per aquelles arestes per les quals ha passat la formiga. El factor d'evaporació es defineix de la següent manera:

$$\tau_{ij} \leftarrow (1-p)\tau_{ij} \tag{3.13}$$

$\forall (i,j) \in L$, on $0 < p <= 1$ és el rati d'evaporació.

Després de l'*evaporació*, totes les formigues deixen una quantitat de feromones a les arestes del seu recorregut de la següent manera:

$$\tau_{ij} \leftarrow \tau_{ij} + \triangle\tau_{ij}^k \tag{3.14}$$

$\forall (i,j) \in L$, on $\triangle\tau_{ij}^k$ es defineix com:

$$\tau_{ij}^k = \begin{cases} \frac{1}{C^k} & \text{si } (i,j) \in T^k \\ 0 & \text{si } (i,j) \notin T^k \end{cases} \tag{3.15}$$

on C^k és el cost del tour T^k construït per la formiga k, En l'Annex 6.2.2 es mostra la seva implementació.

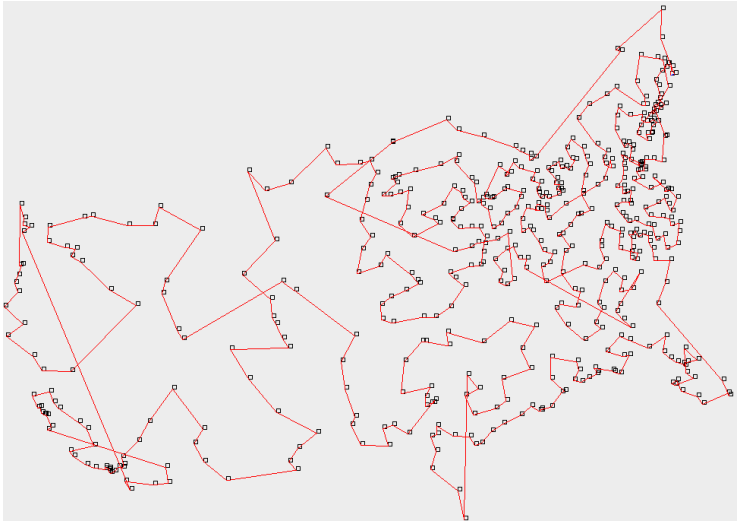

Figura 3.16: Mapa EUA obtingut mitjançant l'ACO- att532

3.4 Resultats obtinguts

Els resultats han sigut fets amb el benchmark TSPLib.

3.4.1 Comparativa

A continuació es llista una taula amb els resultats obtinguts de l'algorisme MST de les instàncies fri26, dantzig42, swiss42, brazil58, att532, pa561 i si1032 del TSPLib [8] . Com a exemple, en la Figura 3.4 es mostra el tour dels EUA trobat per l'algorisme.

Instancia	Millor solució	Valor trobat	GAP(%)	temps(s)
fri26	937.0	1112.0	15.73	0,0030
dantzig42	699.0	933.0	25.08	0,025
swiss42	1273.0	1641.0	22.42	0,0090
brazil58	25395.0	29282.0	13.27	0,025
att532	27686.0	36933.0	25.03	0,174
pa561	2763.0	3928.0	29.65	0,244
si1032	92650.0	97525.0	4.99	0,828

Seguidament es llista una taula amb els resultats obtinguts amb l'algorisme de Lin-Kernighan-Helsgaun. Els resultats han sigut obtinguts mitjançant una llista de nodes candidats de grandària 5. Com a exemple, en la Figura 3.13 es

mostra el tour dels EUA trobat per l'algorisme.

Instancia	Millor solució	Valor trobat	GAP(%)	temps(s)
fri26	937.0	937.0	0	0,084
dantzig42	699.0	699.0	0	0,598
swiss42	1273.0	1273.0	0	1,678
brazil58	25395.0	25459.0	0.25	2,369
att532	27686.0	29483	6.09	142.291
pa561	2763.0	2895.0	4.55	112.220
si1032	92650.0	106050.0	12.63	423.057

El temps s'ha tret d'una execució. Cada execució fa tants trials com número de nodes, excepte si hi ha més de 100 nodes que llavors fa 10 trials.

Finalment, es llista una taula amb els resultats obtinguts de l'algorisme ACO. Com a exemple, en la Figura 3.16 es mostra el tour dels EUA trobat per l'algorisme.[8]

Instancia	Millor solució	Valor trobat	GAP(%)	temps(s)	α	β	p	τ_0	m	C
fri26	937.0	937	0	0,007	1	6	0.5	m/C	536	1112
dantzig42	699.0	736	5.02	1,031	1	4	0.5	m/C	536	933
swiss42	1273.0	1287.0	1.08	1,037	1	6	0.5	m/C	536	1641
brazil58	25395.0	26620.0	4.60	2,156	1	5	0.5	m/C	536	29282
att532	27686.0	32712.0	15.364	435.564	1	6	0.5	m/C	536	36933
pa561	2763.0	3211.0	13.95	610.940	1	6	0.5	m/C	536	3928
si1032	92650.0	106476.0	12.985	1.214.820	1	6	0.5	m/C	536	97525

on τ_0 són les feromones que tenen inicialment les arestes i C és una constant on el seu valor es el cost d'un tour generat a partir d'un MST [2]. El temps d'execució és d'una única execució. Per els problemes de menys de 100 ciutats s'han fet 100 execucions i per als altres tres, 10 execucions.

Observem de les taules que Lin-Kernighan és uns 2 ordres de magnitud més lent que el mst però obtenim millors resultats en general. El joc de proves més gran és l'únic que MST ofereix una solució millor que Lin-Kernighan. Això potser és degut a que mitjançant moviments k-opt i restringint els veïns als 5 amb més *alpha*-nearness l'algorisme cau en un òptim local. Caldria analitzar amb més cura aquest cas ja que no va amb la línia general de millora que proporciona l'algorisme.

Respecte l'ACO s'observa una millora no tan amplia com el Lin-Kernighan però el temps d'execució també creix molt més. Per exemple, l'últim data set LKH triga 7 minuts i l'ACO uns 20 minuts.

Capítol 4

Implementacions paral·leles

4.1 Implementació Paral·lela Lin-Kernighan-Helsgaun

A continuació s'explicarà l'algorisme paral·lel basat en Coherence que s'ha dissenyat en aquest projecte.

En la intro a Coherence (Secció 1.1.3) vam presentar un fragment de codi que ilustra com paral·lelitzar de forma síncrona. En el nostre cas volem que l'execució sigui asíncrona per tal de que a la vegada tots els membres del cluster estiguin executant una iteració de Lin-Kernighan.

La idea es generar de forma aleatòria un tour inicial i enviar aquests tours a l'agent que realitzarà l'optimització. Aquest agent serà enviat a cada membre del cluster i per tant es realitzaran intents (o trials) de forma paral·lela.

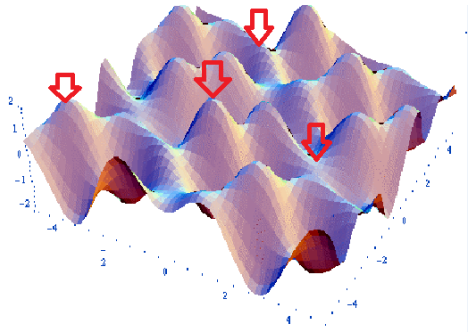

Figura 4.1: Exemple d'una funció amb diferents punts de partida.

`http://plato.asu.edu/gom.html`

D'aquesta manera si en la versió seqüencial cada execució executava una serie de vegades l'algorisme d'optimització, en aquesta versió aquests trials es paral·lelitzen. Cada trial del seqüencial correspon a tants trials del paral·lel com membres del cluster, que intenten trobar un tour millor. D'aquests nous tours trobats només el millor servirà per la generació dels següents trials. A continuació hi ha l'esquelet principal:

Listing 4.1: Esquelet principal de paral·lelització

```
.........
for (int i = 0; i < trial; i += j)
                {
                        ArrayList<Edge> currentTour = null;
                        Member[] members = (Member[]) CacheFactory.getCluster().
                             getMemberSet()
                                        .toArray(new Member[0]);
                        int num = members.length;

                        this.cdl = new CountDownLatch(num);
                        for (j = 0; j < num; ++j)
                        {
                                // Generem un tour inicial a partir d'una ciutat random
                                int n = rand.nextInt(g.getProblemSize());
                                currentTour = this.initialTour(n, new ArrayList<Edge>(),n
                                     );

                                this.getCurrentTourFound(currentTour);
                                this.parallelLinKernighan(currentTour, this.g.
                                     getProblemSize(), j);
                        }
.........

private void parallelLinKernighan(ArrayList<Edge> tour, int size, int iter)
{
...........
        service.setSize(this.g.getProblemSize());
        service.service(tour, iter, this.one_tree, edges, candidates, this.id_edges);
...........
}
```

La versió completa d'aquesta funció la podeu trobar en l'Annex 6.7.

4.2 Implementació Paral·lela Ant Colony

En la versió paral·lelitzada de l'Ant Colony hi ha una diferencia important amb Lin-Kernighan Helsgaun. Mentre que els trials de Lin-Kernighan-Helsgaun són independents, en aquest hi ha d'haver memòria compartida entre les diferents execucions, cosa que ens obliga a usar les possibilitats més potents de Coherence. Per aquest motiu farem servir una cache particionada de tal manera que les feromones es distribuiran a través dels membres del cluster i tots tindran accés als mateixos valors. Un possible escenari d'aquesta arquitectura seria:

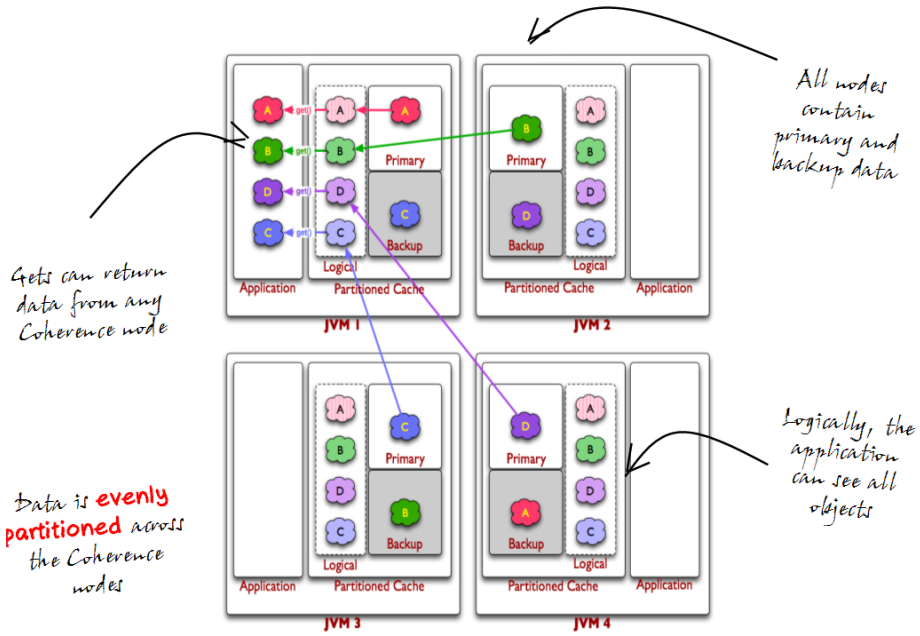

Figura 4.2: Exemple d'un get() a la cache. A, B, D i C són les feromones

Intuitivament el *put()* actualitza el objecte primari i el de backup.

Apart d'aquesta cache també en crearem dues més distribuïdes per als nodes i les arestes. El fet que la tipologia sigui distribuïda es perquè totes les formigues han de tenir una copia però no han d'actualitzar cap informació. Per aquest motiu una vegada s'ha omplert aquesta cache no hi haurà cap valor nou o modificat (no es necessita comunicació).

L'esquelet principal que s'ha utilitzat és el següent:

Listing 4.2: Esquelet principal de paral·lelització

```
.....
for (int i = 0; i < ants; i++) {

                    boolean isMemberValid = false;
                    Member invocationMember = null;

                    while (!isMemberValid) {
                            if (index > members.length - 1)
                                    index = 0;
                            Member m = members[index];
                            if (m.getRoleName().equalsIgnoreCase(
                                    "DedicatedStorageEnabledMember")) {
                                    isMemberValid = true;
                                    invocationMember = m;
                            }
```

```
                }
                HashSet<Member> mS = new HashSet<Member>();
                mS.add(invocationMember);
                iService.execute(new AntInvocable(i), mS, this);
        }
.....
```

I en el *Invocation Observer* escollim el millor tour:

Listing 4.3: Esdeveniment memberCompleted del observer

```
@Override
public void memberCompleted(Member arg0, Object arg1) {
        AntResponse res = (AntResponse) arg1;
        this.cdl.countDown();

        if (res.getDistance() < this.bestPath.getDistance()) {
                System.out.println("Best solution found " + res);
                this.bestPath = res;
                this.canvas.setPath(this.bestPath.getPath());
                this.canvas.repaint();
        }
}
```

4.3 Resultats obtinguts

4.3.1 1-core

Instancia	Millor solució	Valor trobat	GAP(%)	temps(s)
fri26	937.0	937.0	0	2,5
dantzig42	699.0	699.0	0	1,6
swiss42	1273.0	1273.0	0	3,5
brazil58	25395.0	25659.0	1.02	2,800
att532	27686.0	29483	6.09	342.13
pa561	2763.0	2895.0	4.55	410.42
si1032	92650.0	106050.0	12.63	610.210

Aquest augment de temps amb l'ús d'un node és possible que sigui degut a la càrrega de les dades a la cache.

Instancia	Millor solució	Valor trobat	GAP(%)	temps(s)	α	β	p	τ_0	m	C
fri26	937.0	937	0	0,123	1	6	0.5	m/C	536	1112
dantzig42	699.0	699	0	1,119	1	4	0.5	m/C	536	933
swiss42	1273.0	1421.0	10.4	1,065	1	6	0.5	m/C	536	1641
brazil58	25395.0	26981.0	5.87	2,189	1	5	0.5	m/C	536	29282
att532	27686.0	35220	21.39	490.785	1	6	0.5	m/C	536	36933
pa561	2763.0	3332.0	17.07	622.348	1	6	0.5	m/C	536	3928
si1032	92650.0	106310.0	12.84	1.334.876	1	6	0.5	m/C	536	97525

4.3.2 2-core

Instancia	Millor solució	Valor trobat	GAP(%)	temps(s)
fri26	937.0	937.0	0	0,12
dantzig42	699.0	699.0	0	0,9
swiss42	1273.0	1273.0	0	1,2
brazil58	25395.0	25659.0	1.02	1,9
att532	27686.0	29483	6.09	110.321
pa561	2763.0	2895.0	4.55	106.792
si1032	92650.0	106050.0	12.63	370.623

Aquí comencem a observar que el temps és una mica millor que el seqüencial. Observem un speed up en el data set més gran del $S_2 = T_1/T_2 = 423057/370623 = 1,14$ i una eficiencia del $E_2 = S_2/2 = 1,14/2 = 0,57$.

Instancia	Millor solució	Valor trobat	GAP(%)	temps(s)	α	β	p	τ_0	m	C
fri26	937.0	937	0	0,13	1	6	0.5	m/C	536	1112
dantzig42	699.0	699	0	1,2	1	4	0.5	m/C	536	933
swiss42	1273.0	1421.0	10.4	0,956	1	6	0.5	m/C	536	1641
brazil58	25395.0	26981.0	5.87	1,965	1	5	0.5	m/C	536	29282
att532	27686.0	35220	21.39	358.612	1	6	0.5	m/C	536	36933
pa561	2763.0	3332.0	17.07	563.123	1	6	0.5	m/C	536	3928
si1032	92650.0	106310.0	12.84	956.213	1	6	0.5	m/C	536	97525

Pel que fa a l'ACO observem un speed up en el data set més gran del $S_2 = T_1/T_2 = 1214820/956213 = 1,27$ i una eficiencia del $E_2 = S_2/2 = 1,27/2 = 0,63$.

4.3.3 3-core

Instancia	Millor solució	Valor trobat	GAP(%)	temps(s)
fri26	937.0	937.0	0	0,15
dantzig42	699.0	699.0	0	0,8
swiss42	1273.0	1273.0	0	1,1
brazil58	25395.0	25659.0	1.02	1,57
att532	27686.0	29483	6.09	99.632
pa561	2763.0	2895.0	4.55	97.314
si1032	92650.0	106050.0	12.63	312.548

Observem un speed up en el data set més gran del $S_3 = T_1/T_3 = 423057/312548 = 1,35$ i una eficiencia del $E_3 = S_3/3 = 1,35/3 = 0,45$.

Instancia	Millor solució	Valor trobat	GAP(%)	temps(s)	α	β	p	τ_0	m	C
fri26	937.0	937	0	0,15	1	6	0.5	m/C	536	1112
dantzig42	699.0	699	0	0,9	1	4	0.5	m/C	536	933
swiss42	1273.0	1421.0	10.4	0,933	1	6	0.5	m/C	536	1641
brazil58	25395.0	26981.0	5.87	1,562	1	5	0.5	m/C	536	29282
att532	27686.0	35220	21.39	276.432	1	6	0.5	m/C	536	36933
pa561	2763.0	3332.0	17.07	510.746	1	6	0.5	m/C	536	3928
si1032	92650.0	106310.0	12.84	912.388	1	6	0.5	m/C	536	97525

Observem un speed up en el data set més gran del $S_3 = T_1/T_3 = 1214820/912388 = 1,33$ i una eficiencia del $E_3 = S_3/3 = 1,33/3 = 0,44$.

4.3.4 4-core

Instancia	Millor solució	Valor trobat	GAP(%)	temps(s)
fri26	937.0	937.0	0	0,45
dantzig42	699.0	699.0	0	0,9
swiss42	1273.0	1273.0	0	1,02
brazil58	25395.0	25659.0	1.02	1,42
att532	27686.0	29483	6.09	91.36
pa561	2763.0	2895.0	4.55	89.32
si1032	92650.0	106050.0	12.63	293.211

Observem un speed up en el data set més gran del $S_4 = T_1/T_4 = 423057/293211 = 1,44$ i una eficiencia del $E_4 = S_4/4 = 1,44/4 = 0,36$.

Instancia	Millor solució	Valor trobat	GAP(%)	temps(s)	α	β	p	τ_0	m	C
fri26	937.0	937	0	0,23	1	6	0.5	m/C	536	1112
dantzig42	699.0	699	0	0,85	1	4	0.5	m/C	536	933
swiss42	1273.0	1421.0	10.4	0,903	1	6	0.5	m/C	536	1641
brazil58	25395.0	26981.0	5.87	1,128	1	5	0.5	m/C	536	29282
att532	27686.0	35220	21.39	23.306	1	6	0.5	m/C	536	36933
pa561	2763.0	3332.0	17.07	486.326	1	6	0.5	m/C	536	3928
si1032	92650.0	106310.0	12.84	823.487	1	6	0.5	m/C	536	97525

Observem un speed up en el data set més gran del $S_4 = T_1/T_4 = 423057/293211 = 1,47$ i una eficiencia del $E_4 = S_4/4 = 1,44/4 = 0,36$.

Capítol 5

Conclusions

Les conclusions que trec d'aquest treball realitzat és que l'algorisme Lin-Kernighan-Helsgaun sembla ser la millor opció dels tres algorismes amb les dades que hem obtingut en la comparativa 3.4.1, com podíem esperar, en ser un algorisme especialitzat per a TSP en comptes d'AOC, que és molt més genèric.

Un altre factor important que penso que he de mencionar és l'esforç i el guany respecte a implementacions gairebé directes com l'MST. En la meva experiència, la implementació de l'ACO ha sigut força fàcil i intutiva de fer. En canvi, la implementació de LKH ha sigut molt més costosa i requereix entendre conceptes més sofisticats. No obstant, penso que l'esforç realitzat a LKH aporta més benefici, en comparació, que implementar algorismes del estil d'ACO per un problema en concret.

Finalment, concloure que aquest treball és un punt de partida per a qui realment vol resoldre eficientment el problema del TSP. El que s'ha proporcionat aquí són resultats de primeres implementacions i, segurament, són molt més que millorables amb una dedicació més amplia a cadascuna.

5.1 Revisió dels objectius i requisits

Al començament d'aquest treball es van definir una taula de requisits de la solució a fer que podeu veure en la Taula 2.2. Aquests requisits són:

- La solució ha de ser implementada amb Oracle Coherence
- Resol el problema TSP correctament
- Ha de ser una solució escalable i ràpida en donar resultats

El primer punt s'ha aconseguit amb dos de les tres implementacions. La implementació paral·lela del MST no s'ha realitzat donada la complexitat en la bibliografia existent i la manca de temps. El segon punt crec que ha quedat demostrat amb els diferents jocs de proves del benchmark TSPLib. L'últim punt penso que la part de obtenir de forma ràpida els resultats s'ha aconseguit ja que la solució exhaustiva del problema és més lenta en general. Potser implementacions bones sí que són més ràpides però com ja he dit aquestes són unes primeres implementacions, per tant, considerarem que són suficientment ràpides com per haver assolit l'objectiu. El punt d'escalabilitat penso que tan en el cas de l'ACO com LKH sí s'ha aconseguit però s'hauria de trobar formes de paral·lelització alternatives per aprofitar al màxim tècniques de comput paral·lel.

5.2 Treball futur

Ja que hem vist que l'algorisme de Lin-Kernighan-Helsgaun és, segurament, la millor opció per resoldre el TSP, el treball futur seria treballar només en aquest per tal d'aprofitar tot el seu potencial. Per exemple, buscar millores en les estructures de dades utilitzades, etc. Per altre banda també buscaria formes alternatives de paral·lelització que aprofitin més aquestes tècniques.

Part III

Bibliografia

Bibliografia

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: *Introduction to Algorithms*, The MIT Press

[2] Marco Dorigo & Thomas Syützle, *Ant Colony Optimization*, The MIT Press

[3] Keld Helsgaun, *An Effective Implementation of Lin-Kernighan Traveling Salesman Heuristic*

[4] J. D. C. Little, K. G. Murty, D. W. Sweeny & C. Karel, *An algorithm for the traveling salesman problem*, Oper. Res., 11, 972-989 (1963).

[5] M. Grötchel & O. Holland, *Solution of large scale symmetric travelling salesman problems*, Math. Programming, 51, 141-202 (1991).

[6] D. Applegate, R. Bixby, V. Chvàtal & W. Cook, *Finding cuts in the TSP (A preliminary report)*, DIMACS, Tech. Report 95-05 (1995).

[7] Oracle coherence, *Getting Started Guide, Release 3.7.1*, `http://docs.oracle.com/cd/E24290_01/coh.371/e22840.pdf`

[8] TSPLIB 95, *a library of sample instances for the TSP (and related problems) from various sources and of various types*, `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`

[9] Christos H. Papadimitriou. *The Euclidean travelling salesman problem is NP-complete*. Theoretical Computer Science

[10] S. Lin & B. W. Kernighan, "An Effective Heuristic Algorithm for the Traveling-Salesman Problem", Oper. Res. 21, 498-516 (1973).

[11] Stephen Boyd, Lin Xiao, and Almir Mutapcic, *Subgradient Methods*, Notes for EE392o, Stanford University, Autumn, 2003 October 1, 2003

[12] Jon Hall, *Projecte little grid*, `http://littlegrid.bitbucket.org/`

[13] K. Helsgaun. *General k-opt submoves for the Lin-Kernighan TSP heuristic*. Mathematical Programming Computation, 2009.

[14] R.Karp. *Reducibility among combinatorial problems*, in R. E. Miller and J. W. Thatcher (editors). , New York: Plenum. pp. 85-103.

[15] S. Lin & B. W. Kernighan, *An Effective Heuristic Algorithm for the Traveling-Salesman Problem*, Oper. Res. 21, 498-516 (1973).

Part IV

Annexos

Capítol 6

Funcions Java

6.1 Lin-Kernighan Helsgaun

6.1.1 Càlcul de $\beta(i, j)$

Listing 6.1: Càlcul de β

```
private void computeBeta(int godNode)
{
        computeFathers(godNode, godNode);

        for (int j = 0; j < g.getProblemSize(); ++j)
                betaAux.put(j, new double[g.getProblemSize()]);

        this.walkOverTreeBeta(godNode, godNode);


        for (int y = 0; y < g.getProblemSize(); ++y)
        {
                this.beta.put(y, new ArrayList<Double>(this.g.getProblemSize()));
                for (int j = 0; j < g.getProblemSize(); ++j)
                        this.beta.get(y).add(this.betaAux.get(y)[j]);
        }
}

private void computeFathers(int startNode, int currentNode)
{
        for (int son : this.one_tree.get(currentNode))
        {
                if (son == this.excludedNodeOnMST)
                        continue;
                if (!g.gotDad(son) && son != startNode)
                {
                        this.g.addSon(currentNode, son);
                        g.setDad(son, currentNode);
                        computeFathers(startNode, son);
                }
        }
}

private void walkOverTreeBeta(int initialNode, int currentNode)
{
```

```
        this.initialize(currentNode);
        this.visited = new ArrayList<Integer>();
        this.calculateBeta(currentNode);
}

private void initialize(int currentNode)
{
        if (currentNode == this.excludedNodeOnMST)
                return;

        ArrayList<Integer> fills = this.g.getSoon(currentNode);

        for (int i = 0; i < fills.size(); ++i)
        {
                betaAux.get(currentNode)[fills.get(i)] = this.g.getCostOfEdge(currentNode,
                      fills.get(i));
                betaAux.get(fills.get(i))[currentNode] = this.g.getCostOfEdge(currentNode,
                      fills.get(i));
                initialize(fills.get(i));
        }
}

private void calculateBeta(int currentNode)
        {
                ArrayList<Integer> fills = this.g.getSoon(currentNode);
                for (int i = 0; i < this.g.getProblemSize(); ++i)
                {
                        if (currentNode == this.excludedNodeOnMST || i == currentNode ||
                              this.g.getDad(i) == -1)
                                continue;

                        if (this.one_tree.containsKey(currentNode) && this.one_tree.get(
                              currentNode).contains(i))
                        {
                                betaAux.get(currentNode)[i] = this.g.getCostOfEdge(
                                      currentNode, i);
                                betaAux.get(i)[currentNode] = this.g.getCostOfEdge(
                                      currentNode, i);
                        }else if (!fills.contains(i) && !visited.contains(i))
                        {
                                double cost1 = this.betaAux.get(currentNode)[this.g.getDad(i)
                                      ];
                                double cost2 = this.g.getCostOfEdge(i, this.g.getDad(i));
                                double max = (cost1 > cost2) ? cost1 : cost2;
                                betaAux.get(currentNode)[i] = max;
                                betaAux.get(i)[currentNode] = max;
                        }
                }

                for (int i = 0; i < this.g.getProblemSize(); ++i)
                {
                        if (currentNode == this.excludedNodeOnMST || i == currentNode ||
                              this.g.getDad(i) == -1)
                                continue;

                        if (!fills.contains(i) && !visited.contains(i))
                        {
                                visited.add(currentNode);
                                this.calculateBeta(i);
                        }
                }
        }
```

6.1.2 Càlcul subgradient

Listing 6.2: alpha Improve

```
private void alphaImprove()
{
        boolean initialPhase = true;

        boolean firsTime = true;
        double w = -1;
        double t = 2;

        for (int i = 0; i < g.getProblemSize(); ++i)
        {
                pi.put(i, 0.0);
        }

        for (int i = 0; i < g.numberOfEdges(); ++i)
        {
                Edge e = new Edge(g.getIdEdgeOrigin(i), g.getIdEdgeDestination(i)
                        );
                e.setCost(g.getCostEdgeAt(i));
                this.old_cost.add(e);
        }

        do
        {
                this.one_tree = new HashMap<Integer, ArrayList<Integer>>();
                this.compute_1Tree();
                double cost_1tree = this.costOneTree();
                if (firsTime)
                {
                        w = cost_1tree;
                        firsTime = false;
                }

                if (cost_1tree > w)
                {
                        for (int i = 0; i < g.getProblemSize(); ++i)
                                this.best_pi.put(i, this.pi.get(i));

                        w = cost_1tree;
                        if (initialPhase)
                                t *= 2;
                }else if (initialPhase){
                        initialPhase = false;
                        t = 3.0*t/4.0;
                }

                int norm = 0;
                for (int i = 0; i < g.getProblemSize(); ++i)
                {
                        int degree = (this.numberOfTimesInOneTree(i) - 2);
                        this.v_old.put(i, this.v.get(i));
                        this.v.put(i, degree);
                        norm += degree*degree;
                }

                if (norm == 0)
                {
                        this.pi = new HashMap<Integer, Double>();
                        this.g.setPi(this.pi);

                        return;
                }

                t /= 2;
                for (int i = 0; i < g.getProblemSize(); ++i)
                {
```

```
                        double value = this.pi.get(i) + t*(7.0*v.get(i) + 3.0*((this.
                            v_old.get(i) == null) ? this.v.get(i) : this.v_old.get(i))
                            /10.0);

                        this.pi.put(i, value);
                    }

                    this.g.setPi(pi);

                    k++;
                }while(t > 0.0001);
}
```

6.1.3 Tour inicial

Listing 6.3: Funció que retorna el tour inicial

```
private ArrayList<Edge> initialTour(int initial, ArrayList<Edge> toure, int nodeInicio
    )
{
        ArrayList<Edge> tour = this.copyTour(toure);
        if (numberOfTimesInTour(tour, initial) >= 2)
                return null;

        if (tour.size() == g.getProblemSize() - 1)
        {
                int j = nodeInicio;
                int origen;
                int destino;
                if (initial > j)
                {
                        destino = initial;
                        origen = j;
                }else{
                        destino = j;
                        origen = initial;
                }
                int id = g.getIdEdge(origen, destino);
                Edge e = new Edge(origen, destino);
                e.setEdgeId(id);
                tour.add(e);

                if (this.isTour(tour, null))
                        return tour;
                else
                {
                        return null;
                }


        }

        for (int i = 0; i < g.getNumberOfCandidates(initial); ++i)
        {
                int j = g.getCandidateAt(initial, i);

                if (numberOfTimesInTour(tour, j) >= 2)
                        continue;

                int origen;
                int destino;
                if (initial > j)
                {
                        destino = initial;
                        origen = j;
```

```
            }else{
                    destino = j;
                    origen = initial;
            }
            int id = g.getIdEdge(origen, destino);

            if (this.inTour(initial, j, tour))
                    continue;

            if (g.getAlpha(id) == 0)
            {
                    if (this.inBesttour(initial, j))
                    {
                            Edge e = new Edge(origen, destino);
                            e.setEdgeId(id);
                            tour.add(e);
                            ArrayList<Edge> aux = initialTour(j, tour, nodeInicio);
                            if (aux == null)
                            {
                                    tour.remove(tour.size() - 1);
                            }else return aux;
                    }

            }
    }

    for (int i = 0; i < g.getNumberOfCandidates(initial); ++i)
    {
            int j = g.getCandidateAt(initial, i);

            if (numberOfTimesInTour(tour, j) >= 2)
                    continue;

            int origen;
            int destino;
            if (initial > j)
            {
                    destino = initial;
                    origen = j;
            }else{
                    destino = j;
                    origen = initial;
            }
            int id = g.getIdEdge(origen, destino);

            if (this.inTour(initial, j, tour))
                    continue;

            Edge e = new Edge(origen, destino);
            e.setEdgeId(id);
            tour.add(e);

            ArrayList<Edge> aux = initialTour(j, tour, nodeInicio);
            if (aux == null)
            {
                    tour.remove(tour.size() - 1);
            }else return aux;
    }

    for (int i = 0; i < g.getProblemSize(); ++i)
    {
            if (i == initial)
                    continue;

            int j = i;
            if (numberOfTimesInTour(tour, j) >= 2)
                    continue;
```

```
                int origen;
                int destino;
                if (initial > j)
                {
                        destino = initial;
                        origen = j;
                }else{
                        destino = j;
                        origen = initial;
                }

                if (this.inTour(initial, j, tour))
                        continue;

                int id = g.getIdEdge(origen, destino);
                Edge e = new Edge(origen, destino);
                e.setEdgeId(id);
                tour.add(e);


                ArrayList<Edge> aux = initialTour(j, tour, nodeInicio);
                if (aux == null)
                        tour.remove(tour.size() - 1);
                else return aux;
        }

        System.out.println("ERROR generating initial tour");
        return null;
}
```

6.1.4 Determinar si un r-move genera un tour

Listing 6.4: Funció que determina si un canvi genera un tour o no

```
private boolean isTour3Opt(int t1, int t2, int t3, int t4, int t5, int t6)
        {
                // tX són les posicions corresponents en el array que representa el tour
                int t2_vis = t2;
                int t3_vis = t3;
                int t4_vis = t4;

                boolean is2OptTour = isTour2Opt(t1, t2, t3, t4);
                if (is2OptTour)
                {
                        t2 = t4;
                        t3 = t5;
                        t4 = t6;

                        if (t2_vis == t5)
                                t3 = t2;
                        if (t2_vis == t6)
                                t4 = t2;

                        if (t3_vis == t4_vis)
                                t2 = t3;
                        if (t3_vis == t6)
                                t4 = t3;

                        if (t4_vis == t5)
                                t3 = t4;

                        return isTour2Opt(t1, t2, t3, t4);
                }else{
                        return isBetweenNotBasic(t1, t2, t3, t5);
                }
```

```
}

private boolean isTour2Opt(int t1, int t2, int t3, int t4)
{
        return this.isBetweenNotBasic(t1, t2, t3, t4);
}

private boolean isBetweenNotBasic(int t1_pos, int t2_pos, int t3_pos, int t5_pos)
{
        if (t2_pos <= t5_pos && t5_pos <= t3_pos && t3_pos <= t1_pos)
                return true;

        if (t1_pos <= t3_pos && t3_pos <= t5_pos && t5_pos <= t2_pos)
                return true;

        if (t3_pos <= t1_pos && t1_pos <= t2_pos && t2_pos <= t5_pos)
                return true;

        if (t5_pos <= t2_pos && t2_pos <= t1_pos && t1_pos <= t3_pos)
                return true;

        if (t3_pos <= t5_pos && t5_pos <= t2_pos && t2_pos <= t1_pos)
                return true;

        if (t5_pos <= t3_pos && t3_pos <= t1_pos && t1_pos <= t2_pos)
                return true;

        if (t1_pos <= t2_pos && t2_pos <= t5_pos && t5_pos <= t3_pos)
                return true;

        if (t2_pos <= t1_pos && t1_pos <= t3_pos && t3_pos <= t5_pos)
                return true;

        return false;
}
```

6.2 Ant colony

6.2.1 Sel·lecció de la següent ciutat (roulette wheel selection)

Listing 6.5: Funció que calcula la probabilitat d'anar a les diferents ciutats

```
public int calculateMovementsProbability(
                HashMap<VertexKey, Vertex> vertex,
                HashMap<EdgeKey, Pheromone> pheromones, HashMap<EdgeKey, Edge> edges)
                    {

        Random rnd = new Random();
        double[] probabilities = new double[vertex.size()];
        double[] probabilitiesP = new double[vertex.size()];
        double total = 0;

        int currentVertex = this.path.get(this.path.size()-1).getId();

        for (int i = 0; i < probabilities.length; i++) {

                if (i == currentVertex || this.path.contains(new VertexKey(i)))
                        probabilities[i] = 0;
                else {
                        probabilities[i] = Math.pow(
                                        MapsUtils.getPheromone(currentVertex, i,
                                            pheromones)
```

```
                                                .getPheromones(), Constants.ALPHA)
                                        * Math.pow(
                                                (1f / MapsUtils.getEdge(
                                                     currentVertex, i,
                                                               vertex, edges).
                                                                    getDistance()),
                                                Constants.BETA);
                }
                total += probabilities[i];
        }

        for (int i = 0; i < probabilities.length; i++) {
                probabilitiesP[i] = probabilities[i] / total;
        }

        double[] cumulative = new double[probabilities.length + 1];
        Arrays.fill(cumulative, 0);
        for (int i = 0; i < probabilitiesP.length; ++i)
                cumulative[i + 1] = cumulative[i] + probabilitiesP[i];

        double p = rnd.nextDouble();

        for (int i = 0; i < cumulative.length-1; ++i)
                if (p>=cumulative[i] && p<cumulative[i+1])
                        return i;

        return -1;
}
```

6.2.2 Actualització de les feromones

Listing 6.6: Actualització de les feromones

```
public void updatePheromones(HashMap<VertexKey, Vertex> vertex,
                HashMap<EdgeKey, Pheromone> pheromones, HashMap<EdgeKey, Edge> edges)
                     {

        for (int i=0;i<vertex.size();i++) {
                for (int j=i+1;j<vertex.size();j++) {
                        double decrease = (1.0-Constants.PEC)*MapsUtils.getPheromone(i, j
                             , pheromones).getPheromones();
                        double increase = 0;

                        if (this.edgeOnPath(i, j, vertex, pheromones, edges)) {
                                increase = (Constants.Q/this.getPathDistance(vertex,edges)
                                     );
                        }

                        MapsUtils.setPheromone(i, j, (decrease+increase), pheromones);
                }
        }
```

6.3 Versió paral·lela Lin-Kernighan Helsgaun

6.3.1 Esquelet principal

Listing 6.7: Esquelet principal de paral·lelització

```
private ArrayList<Edge> findTour()
{
```

```
        int trial = (this.g.getProblemSize() > 100) ? 10 : this.g.getProblemSize();
        double betterCost = -1;
        double currentCost;
        ArrayList<Edge> currentBestTour = new ArrayList<Edge>();

        this.getEdges();
        int j = 0;
        for (int i = 0; i < trial; i += j)
        {
                ArrayList<Edge> currentTour = null;
                Member[] members = (Member[]) CacheFactory.getCluster().getMemberSet()
                                .toArray(new Member[0]);
                int num = members.length;

                this.cdl = new CountDownLatch(num);
                for (j = 0; j < num; ++j)
                {
                        int n = rand.nextInt(g.getProblemSize());
                        currentTour = this.initialTour(n, new ArrayList<Edge>(),n);
                        this.getCurrentTourFound(currentTour);
                        this.parallelLinKernighan(currentTour, this.g.getProblemSize(), j);
                }

                try {
                        this.cdl.await();
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
                        currentTour = this.retrieveSolution();
                        if (currentTour != null)
                        {
                                currentCost = this.currentCost(currentTour);

                                if (currentCost < betterCost || betterCost == -1)
                                {
                                        betterCost = currentCost;
                                        currentBestTour = this.copyTour(currentTour);
                                        this.adjustCandidateSet(currentTour);
                                        this.bestTour = this.copyTour(currentBestTour);
                                        NamedCache bestT = CacheFactory.getCache("Tour");
                                        bestT.put(0, this.bestTour);
                                }
                        }
        }


        this.resetCandidateSet();

        return currentBestTour;
}
```