; revised .

Algorismes per al problema del viatjant de comerç: estudi de mètodes aproximats, concurrents i eficients

Dylan Canning Garcia[1], Antonio Marí González[2], Juan Marí González[3], Antoni Jaume Lemesev [4]

[1]Estudiant Enginyeria Informàtica UIB, 49608104W
[2]Estudiant Enginyeria Informàtica UIB, 46390336A
[3]Estudiant Enginyeria Informàtica UIB, 46390338M
[4]Estudiant Enginyeria Informàtica UIB, 43223871V

Alumne d'entrega: Dylan Canning (email: dcg750@id.uib.eu).

ABSTRACT Aquest document presenta el disseny i la implementació d'una aplicació MVC desenvolupada en Java 23 i Swing per a la resolució i visualització del problema del viatjant de comerç (Travelling Salesman Problem, TSP). S'implementen cinc algorismes: Força Bruta, Programació Dinàmica (Held-Karp), Branch and Bound, Branch and Bound Concurrent i Nearest Neighbor (Greedy).
El desenvolupament s'ha realitzat en el marc de l'assignatura d'Algorismes Avançats. Es detallen les tècniques emprades, com la definició de classes, l'enfocament de programació dinàmica, la programació concurrent ,tot integrat en una arquitectura MVC.
Finalment, el document conclou detallant els diferents problemes trobats amb la implementació concurrent del Branch and Bound, comentant possibles zones problemàtiques del codi concurrent.

I. Introducció

AQUESTA pràctica s'ha desenvolupat seguint una metodologia estructurada basada en l'arquitectura Model-Vista-Controlador (MVC), tal com s'ha treballat a classe. Aquesta estructura modular ens ha permès dividir el treball en paquets independents, facilitant l'organització i agilitzant el desenvolupament. A partir de l'experiència de pràctiques anteriors, hem modularitzat una mica més la part del MVC.

La distribució de tasques dins el nostre equip de quatre membres s'ha fet seguint els components propis del patró MVC [10]: un membre s'ha encarregat de la implementació de la interfície gràfica (Vista), dos membres han desenvolupat la lògica de dades i els algorismes (Model), i un altre ha implementat el controlador que coordina la interacció entre els components anteriorment esmentats. A més, hem continuat utilitzant el patró "Factory Methodçom hem fet als esquemes anteriors.

En aquest projecte, el patró "Factory Method"s'ha implementat mitjançant la classe especialitzada `AlgorithmFactory`, que encapsula la lògica d'instanciació dels diferents algorismes TSP. Això permet que la resta del codi no hagi de conèixer els detalls de creació dels objectes, ja que aquesta responsabilitat recau en mètodes específics que, a partir del paràmetre `AlgorithmType`, generen dinàmicament la instància adequada. Aquesta separació afavoreix la modularitat i facilita la incorporació de nous algorismes al projecte de manera dinàmica i senzilla.

El paràmetre `AlgorithmType` és una enumeració que defineix de manera centralitzada tots els tipus d'algorismes disponibles per al càlcul de rutes, permetent una selecció i comparativa més clara i mantenible entre ells. Aquestes dues definicions han contribuït a una metodologia de treball més àgil i independent.

Per tal de facilitar la col·laboració i el seguiment del projecte, hem continuat utilitzant l'entorn emprat a les múltiples pràctiques anteriors. És a dir, l'ús d'un repositori de GitHub per al control de versions, que ens permet anar documentant i compartint els avanços de manera sistemàtica. També hem fet ús de l'eina *Obsidian* per a la creació de diagrames i la creació de relacions entre idees i arxius. A més, hem mantingut l'ús de l'IDE IntelliJ IDEA per mantenir la compatibilitat dels formats de projecte.

Amb aquestes bases metodològiques establertes, hem procedit a definir els aspectes tècnics específics de cada secció (algorismes, estructures de dades, processos, etc.) necessaris per a la implementació. L'objectiu central del disseny ha estat aplicar i maximitzar els coneixements adquirits a l'assigna-

tura, especialment pel que fa a estratègies de Programació Dinàmica, Concurrència i Disseny Modular, que han estat fonamentals per a la implementació fàcil i eficient dels diferents algorismes investigats.

II. Metodologia de feina

Per a aquest projecte, hem seguit una metodologia estructurada i col·laborativa que ens ha facilitat integrar diverses eines i tècniques per obtenir un producte final coherent, sense necessitat de realitzar sessions presencials intensives amb tots els companys. En aquest apartat, explicarem els aspectes relacionats amb l'entorn de programació i el sistema de control de versions que hem utilitzat durant tot el procés de desenvolupament, així com els canvis respecte a la primera pràctica.

Per aquest projecte hem seguint amb una metodologia estructurada i col·laborativa, que ens ha permès integrar diferents eines i tècniques per aconseguir un producte final coherent sense haver de fer intensius presencials amb tots els companys. En aquest apartat detallarem els aspectes relacionats amb l'entorn de programació i el sistema de control de versions que hem emprat al llarg de tot el procés de desenvolupament, així com els canvis respecte a la primera pràctica.

A. Entorn de Programació

El desenvolupament de l'aplicació l'hem dut a terme utilitzant l'IDE *IntelliJ IDEA*, que ens ha proporcionat un entorn robust i modern per a programar en Java, bastant més amigable que *NetBeans*. S'ha treballat amb la versió *Java 23*, aprofitant les seves millores i estabilitat per garantir una execució eficient dels càlculs i processos implementats, com també la facilitat de maneig amb objectes gràfics.

Per a aquesta pràctica, hem fet ús de JavaFX per a la interfície gràfica d'usuari per mostrar l'arbre filogenètic i el graf. En concret, hem utilitzat els diversos paquets que ofereix JavaFX per a la representació d'escenes, permetent realitzar els gràfics corresponents en dues i tres dimensions. A més, hem incorporat la interacció amb l'usuari per tal de facilitar la manipulació i exploració dels núvols de punts de manera intuïtiva i dinàmica.

B. Control de Versions

Hem seguit amb l'estructura que varem implementar a la segona pràctica degut a que va funcionar molt bé, es a dir centralitzat al repositori principal on s'allotgen les dues practiques anteriors però seguint amb el nostre sistema de control de versions basat en **Git**, amb el repositori centralitzat allotjat a **GitHub**. Aquesta estratègia ha facilitat:

- Mantenir un historial detallat de totes les modificacions realitzades al codi font.
- Coordinar el treball col·laboratiu entre els membres de l'equip, permetent la integració de diferents funcionalitats de manera ordenada.
- Assegurar una revisió contínua del codi i una resolució ràpida dels conflictes que sorgeixin durant el desenvolupament.

La utilització de GitHub es clau per a la nostra gestió de les versions, degut que ens garanteix que el projecte romangui coherent i actualitzat en tot moment, cosa que afavoreix molt el desenvolupament. A més de l'esmentat, la IDE emprada ens ha permès dur a terme aquest control de versions de manera senzilla pels integrants que no han emprat abans la feina, ja que posseeix eines de gestió de versions sense haver d'utilitzar la terminal.

III. Fonaments Teòrics

En aquesta secció tractarem els diversos elements teòrics que hem emprat per a la nostra pràctica.

A. Programació Dinámica

En l'àmbit de la informàtica, molts problemes d'optimització i anàlisi de dades complexes presenten solucions recursives que, si s'implementen de manera directa, resulten computacionalment ineficients. La programació dinàmica (PD) és una tècnica poderosa per abordar aquests reptes, especialment quan les subsolucions d'un problema se solapen, permetent emmagatzemar resultats intermedis i evitar càlculs redundants. Aquesta tècnica es fonamenta en el principi d'optimalitat de Bellman, que estableix que tota subseqüència d'una solució òptima també ha de ser òptima.

En el cas del Viatjant de Comerç (TSP), la programació dinàmica es pot utilitzar per resoldre el problema de manera més eficient. L'algorisme de Held-Karp que hem implementat aplica Programació Dinàmica per calcular el camí mínim que visita totes les ciutats una sola vegada i retorna a l'origen. Aquest algorisme emmagatzema les distàncies mínimes per a cada subconjunt de ciutats i cada possible ciutat final, reduint dràsticament el nombre de càlculs necessaris respecte a una solució recursiva pura. Tot i ser més eficient que un Branch and Bound o cerca exhaustiva (factorials), cal destacar que és exponencial.

En aquesta pràctica, s'ha desenvolupat una aplicació amb interfície gràfica d'usuari (GUI), estructurada segons el patró Model-Vista-Controlador (MVC), que permet generar, carregar i resoldre instàncies del TSP. L'objectiu és trobar el recorregut òptim utilitzant tècniques com Branch and Bound, però la programació dinàmica també es pot aplicar per a instàncies de mida moderada, oferint una alternativa eficient.

Aquest enfocament no només permet respondre preguntes com ara "Quin és el camí més curt que visita totes les ciutats?", sinó que també facilita l'anàlisi de l'eficiència dels diferents algoritmes, la visualització del graf i de la solució òptima, i l'extensió de l'aplicació a altres variants del problema, com el TSP asimètric o amb restriccions addicionals.

B. Patró Factory Method

El patró *Factory Method* és un disseny de creació que hem decidit utilitzar perquè permet delegar la creació d'objectes a subclasses, facilitant així la modularitat a l'hora d'afegir nous algorismes per resoldre el TSP. En el context d'aquesta aplicació, hem implementat aquest patró mitjançant la classe `AlgorithmFactory`, que s'encarrega de generar instàncies dels diferents algorismes de TSP segons els paràmetres proporcionats.

Tots els algorismes implementen la interfície comuna `TSPAlgorithm`, que defineix el contracte que han de seguir totes les implementacions d'algorismes de TSP.

A continuació, es mostra el diagrama de classes que exemplifica aquesta arquitectura:

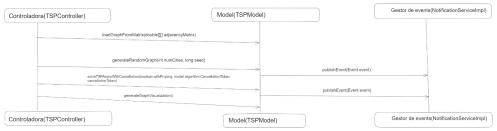

Figura 1. **Model-Controladora**

C. Model

La classe AlgorithmFactory és responsable de crear instàncies dels diferents algorismes de resolució del TSP disponibles al projecte. Aquesta classe utilitza el patró de disseny "Factory Method"per encapsular la lògica de creació, permetent que el codi client només necessiti especificar el tipus d'algorisme que vol utilitzar, sense preocupar-se pels detalls d'implementació.

Pel que fa als algorismes, el tipus Brute Force calcula la ruta òptima enumerant totes les permutacions possibles de ciutats, garantint la solució exacta però amb una complexitat exponencial. L'algorisme Held-Karp utilitza programació dinàmica amb bitmasks per reduir la complexitat temporal i espacial, sent més eficient per a un nombre moderat de ciutats. Altres algorismes, com Branch and Bound o heurístiques, poden estar disponibles per equilibrar precisió i eficiència segons la mida del problema.

Pel que fa als algorismes, el tipus Branch and Bound és el mètode principal implementat per resoldre el TSP en aquest projecte. Explora l'espai de solucions del TSP de manera sistemàtica, descartant rutes que no poden millorar la millor solució trobada fins al moment mitjançant tècniques de poda. Això permet trobar la solució òptima sense haver d'examinar totes les permutacions possibles.

L'algorisme de força bruta examina totes les permutacions possibles de les ciutats per trobar la ruta més curta. Garanteix la solució òptima, però la seva complexitat factorial el fa inviable per a més d'unes poques ciutats.

L'algorisme Greedy selecciona sempre la ciutat no visitada més propera en cada pas. És molt ràpid i senzill, però no garanteix trobar la solució òptima, ja que pot quedar atrapat en mínims locals.

Finalment, l'algorisme Held-Karp utilitza programació dinàmica amb bitmasks per resoldre el TSP de manera exacta i molt més eficient que la força bruta. Redueix dràsticament el nombre de càlculs necessaris, però segueix sent limitat a instàncies de mida mitjana.

La classe TSPModel actua com el nucli del model en l'arquitectura del projecte, integrant i gestionant els diferents components relacionats amb la resolució del TSP. És responsable de coordinar l'ús dels algorismes, la matriu de distàncies i la generació d'estructures derivades com els grafs i les estadístiques d'execució.

Pel que fa als algorismes, TSPModel permet seleccionar i aplicar diferents mètodes de resolució, com Branch and Bound, Brute Force, Held-Karp, entre d'altres. Aquests algorismes s'utilitzen per calcular la ruta òptima o aproximada entre ciutats, i els resultats s'emmagatzemen en una instància de TSPSolution, que és gestionada directament per aquesta classe.

A més, TSPModel centralitza la lògica d'esdeveniments i notificacions, publicant actualitzacions sobre el progrés del càlcul, errors o la disponibilitat de dades com la millor ruta trobada o les estadístiques d'execució. Això la converteix en la classe que "junta tot"en el model, connectant els algorismes, les dades i les notificacions perquè la resta de l'aplicació pugui interactuar amb ells de manera estructurada.

El procés de solució del TSP inclou diverses etapes, com la validació del format de les dades per assegurar que només es puguin carregar matrius d'adjacència vàlides. Això permet evitar errors durant l'execució del programa. Un cop carregades, les ciutats i les distàncies es guarden en estructures de dades adequades per facilitar la seva manipulació i ús en els càlculs.

D. Vista

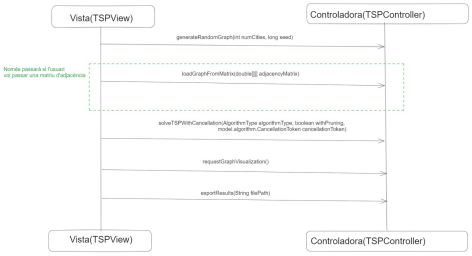

Figura 2. **Vista-Controladora**

La classe TSPView és responsable de representar visualment el problema del viatjant de comerç (TSP) mitjançant una interfície gràfica basada en Swing. Aquesta classe organitza i mostra els diferents components de la interfície, com panells de control, àrees de visualització del graf, estadístiques i resultats. Permet a l'usuari generar o carregar grafs (mitjançant una matriu d'adjacència), seleccionar l'algorisme de resolució, iniciar o aturar l'execució i visualitzar tant la matriu d'adjacència com la solució òptima trobada. També gestiona la visualització gràfica del graf i del camí òptim,

assegurant que la informació es presenti de manera clara i llegible.

Més específicament, la classe interna GraphCanvas (dins TSPView) s'encarrega de dibuixar el graf i la solució òptima dins de la interfície Swing. Utilitza les dades del model per representar visualment les ciutats i les connexions entre elles, destacant el camí òptim trobat per l'algorisme. Aquesta classe pot adaptar la visualització segons les dades rebudes, mostrant tant el graf inicial com la solució final, i permetent a l'usuari explorar visualment el resultat del càlcul.

E. Controladora i Notificacions

La classe TSPController és l'encarregada de coordinar la interacció entre el model (TSPModel) i la vista (TSPView) dins l'arquitectura de l'aplicació. Actua com a intermediari gestionant la lògica de l'aplicació, inicialitzant les dependències necessàries i responent a les accions de l'usuari, com carregar o generar grafs, executar l'algoritme TSP, o exportar resultats. Entre els seus mètodes més importants es troben el constructor TSPController(TSPModel model, TSPView view, NotificationService notificationService), que configura les connexions entre els components, i handleNotification(Event event), que processa les notificacions rebudes de la interfície d'usuari o d'altres components.

El mètode handleNotification(Event event) és fonamental per gestionar les notificacions generades pel sistema o altres parts de l'aplicació. Aquest mètode utilitza un "switch"per identificar el tipus d'esdeveniment (informatiu, advertència o error) i executar l'acció corresponent. Això assegura una comunicació fluida i reactiva entre els components de l'aplicació.

El NotificationService és una interfície que defineix el mecanisme de publicació i subscripció d'esdeveniments dins l'aplicació, facilitant la comunicació entre els diferents components. Els seus mètodes principals són publishEvent(Event event), que permet publicar un esdeveniment perquè altres components el rebin, i subscribe(EventType type, EventListener listener), que registra un observador per a un tipus d'esdeveniment específic.

La classe NotificationServiceImpl és la implementació concreta de NotificationService. Gestiona la distribució d'esdeveniments utilitzant una estructura interna per emmagatzemar els observadors registrats. Els seus mètodes clau inclouen publishEvent(Event event), que notifica tots els observadors subscrits a un tipus d'esdeveniment, i unsubscribe(EventType type, EventListener listener), que elimina un observador registrat. Aquesta classe assegura una comunicació fluida i desacoblada entre els components del programa.

F. Flux d'Esdeveniments de l'Aplicació

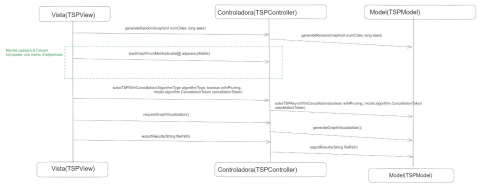

Figura 3. Vista Grosa de la interacció

El funcionament intern del sistema es basa en un model d'interacció orientat a esdeveniments, estructurant la comunicació entre els diferents components mitjançant un patró de control centralitzat. En l'Annex 4, es presenta un diagrama de seqüència que il·lustra el flux complet d'esdeveniments durant l'execució de la resolució del problema del viatjant (TSP) dins de l'aplicació.

El flux comença amb l'arrencada de l'aplicació, on l'usuari interactua amb la interfície gràfica (Vista). Aquesta instancia el controlador principal (TSPController) i registra els canals de notificació asíncrona a través del servei de notificacions (NotificationServiceImpl). L'usuari introdueix els paràmetres d'execució —com el el tipus d'algorisme a emprar, els paràmetres per generar un graf aleatori, o una matriu d'adjacència personalitzada— mitjançant els elements interactius de la Vista. En prémer el botó de resolució ("Resoldre TSP"), el controlador valida els paràmetres i llança la tasca de càlcul en segon pla per mantenir la interfície fluida.

El càlcul de les distàncies de Levenshtein (o els altres algorismes seleccionables) es realitza sobre les dades proporcionades, i les notificacions de progrés es propaguen cap a la Vista mitjançant el servei de notificacions. Això permet actualitzar la interfície en temps real, mostrant informació contextual de l'estat actual, com és la barra de progrés.

La resolució del TSP (amb l'algorisme seleccionat) es realitza sobre el graf actual, i les notificacions de progrés es propaguen cap a la Vista mitjançant el servei de notificacions. Això permet actualitzar la interfície en temps real, mostrant informació contextual de l'estat actual, com la barra de progrés, el nombre de nodes explorats o podats, i la cota actual.

Un cop finalitzat el càlcul, el controlador sol·licita la visualització del graf i la solució trobada. El resultat es notifica novament a la Vista, que s'encarrega de mostrar gràficament la ruta òptima, ressaltar els nodes rellevants i habilitar funcionalitats com l'exportació de resultats.

La interacció conclou amb la possibilitat d'explorar la solució generada o consultar detalls addicionals sobre el càlcul. Durant tot el procés, el sistema incorpora mecanismes de gestió d'errors: si es detecta algun problema durant el càlcul, la generació del graf o la visualització, es comunica immediatament a la Vista mitjançant el sistema de notificacions, que actualitza l'estat de la interfície i informa l'usuari de l'incident.

Aquest enfocament modular i basat en esdeveniments permet mantenir una separació clara de responsabilitats entre les capes de l'aplicació, assegurant una resposta àgil i fluida a les accions de l'usuari.

IV. Algorismes implementats, optimitzacions i paral·lelisme

Aquesta secció analitza en profunditat els algorismes implementats per a la resolució del Problema del Viatjant de Comerç (TSP). S'examinen quatre enfocaments principals: Força Bruta, l'heurística Voraç (Greedy), l'algorisme exacte de Held-Karp basat en programació dinàmica, i l'estratègia de Ramificació i Poda (Branch and Bound), incloent-hi una variant seqüencial i una d'alt rendiment concurrent. Per a cada mètode, es detalla la lògica subjacent, el pseudocodi, una anàlisi de complexitat computacional i espacial, i una discussió sobre les tècniques de paral·lelisme aplicades per a optimitzar-ne l'execució.

A. Algorisme de Força Bruta (Brute Force)

L'algorisme de Força Bruta és el mètode més directe per resoldre el TSP. Garanteix la troballa de la solució òptima absoluta, ja que explora de manera exhaustiva totes les rutes possibles. La seva lògica es basa a generar cada permutació de les ciutats, calcular el cost total de la ruta corresponent i retenir aquella amb el cost mínim.

La implementació (BruteForceTSP.java) fixa una ciutat d'inici (per exemple, la ciutat 0) i genera totes les permutacions possibles de les $n-1$ ciutats restants. Per a cada permutació, construeix un cicle complet que comença i acaba a la ciutat inicial, n'avalua el cost sumant les distàncies entre ciutats consecutives i actualitza la millor solució trobada fins al moment. Donada la seva naturalesa combinatòria, aquest mètode només és pràctic per a un nombre molt reduït de ciutats (típicament $n \leq 10$), tal com s'adverteix al codi (MAX_PRACTICAL_SIZE = 10).

a: Anàlisi de Complexitat:

- **Complexitat Temporal:** La generació de totes les permutacions de $n-1$ ciutats té un cost de $(n-1)!$. Per a cada permutació, el càlcul del cost del recorregut requereix n sumes. Per tant, la complexitat total és $O(n \cdot (n-1)!) = O(n!)$. Aquest creixement factorial fa que l'algorisme sigui intractable per a valors de n moderats.
- **Complexitat Espacial:** La memòria principal requerida és per a emmagatzemar la permutació actual i el millor camí trobat, la qual cosa resulta en una complexitat de $O(n)$. Si es pre-generen totes les permutacions, l'espai seria $O(n \cdot n!)$, però la implementació (generatePermutationsRecursive) les genera iterativament, mantenint el cost espacial baix.

b: Paral·lelisme:

La implementació proporcionada de Força Bruta és purament seqüencial. Tot i que el problema és inherentment paral·lelitzable (cada permutació es podria avaluar en un fil separat), no s'ha implementat aquesta estratègia, ja que el seu ús pràctic està limitat a instàncies tan petites que el guany de la paral·lelització seria negligible.

Algorithm 1 Algorisme de Força Bruta per al TSP

Require: Matriu de distàncies D, nombre de ciutats n
Ensure: Solució òptima (C_{min}, P_{opt})

$P \leftarrow$ llista de ciutats $[1, 2, \ldots, n-1]$
$Permutacions \leftarrow$ generar totes les permutacions de P
$C_{min} \leftarrow \infty$
$P_{opt} \leftarrow$ null
for cada permutació p a $Permutacions$ **do**
    $RutaActual \leftarrow [0] + p + [0]$ ▷ Construeix el cicle complet
    $CostActual \leftarrow 0$
    for $i = 0$ to $n-1$ **do**
        $CostActual \leftarrow CostActual + D[RutaActual[i]][RutaActual[i+1]]$
    end for
    if $CostActual <; C_{min}$ **then**
        $C_{min} \leftarrow CostActual$
        $P_{opt} \leftarrow RutaActual$
    end if
end for
return (C_{min}, P_{opt})

B. Algorisme Voraç (Greedy TSP)

L'algorisme Voraç, conegut com l'heurística del "veí més proper"(Nearest Neighbor), construeix una solució al TSP de manera incremental i ràpida. A cada pas, pren la decisió localment òptima, que consisteix a viatjar a la ciutat no visitada més propera. Tot i que aquesta estratègia no garanteix la solució òptima global, sol proporcionar resultats raonablement bons en un temps molt reduït.

La implementació (GreedyTSP.java) parteix d'una ciutat inicial. En cada etapa, cerca entre totes les ciutats no visitades aquella que es troba a la mínima distància de la ciutat actual. Aquesta ciutat esdevé la següent en el camí i es marca com a visitada. El procés es repeteix fins que totes les ciutats han estat visitades, moment en què es tanca el cicle tornant a la ciutat d'origen.

a: Anàlisi de Complexitat:

- **Complexitat Temporal:** Per a cadascuna de les $n-1$ etapes de construcció del camí, l'algorisme ha de buscar la ciutat més propera entre les restants, la qual cosa implica una cerca lineal sobre fins a $n-1$ ciutats. Això resulta en una complexitat temporal de $O(n^2)$.
- **Complexitat Espacial:** Es necessita espai per emmagatzemar el camí construït i un vector booleà per al seguiment de les ciutats visitades. Ambdós requereixen un espai de $O(n)$.

Algorithm 2 Algorisme Voraç del Veí Més Proper

Require: Matriu de distàncies D, ciutat inicial c_0
Ensure: Solució aproximada (C_{total}, P)
$n \leftarrow$ nombre de ciutats
$P \leftarrow [c_0]$
$Visitades[c_0] \leftarrow$ true
$C_{actual} \leftarrow c_0$
$C_{total} \leftarrow 0$
for $i = 1$ to $n-1$ **do**
  $C_{propera} \leftarrow -1$
  $D_{min} \leftarrow \infty$
  for $j = 0$ to $n-1$ **do**
    if not $Visitades[j]$ and $D[C_{actual}][j] < D_{min}$ **then**
      $D_{min} \leftarrow D[C_{actual}][j]$
      $C_{propera} \leftarrow j$
    end if
  end for
  $P.afegir(C_{propera})$
  $Visitades[C_{propera}] \leftarrow$ true
  $C_{total} \leftarrow C_{total} + D_{min}$
  $C_{actual} \leftarrow C_{propera}$
end for
$C_{total} \leftarrow C_{total} + D[C_{actual}][c_0]$ ▷ Tornada a l'origen
$P.afegir(c_0)$
return (C_{total}, P)

b: Paral·lelisme:

La qualitat de la solució voraç depèn fortament de la ciutat de partida. Per mitigar aquesta dependència, la implementació GreedyTSP.java inclou un mètode paral·lel (solveParallelGreedy). Aquesta versió executa l'algorisme del veí més proper simultàniament des de múltiples ciutats inicials. Utilitza un ExecutorService amb un nombre de fils igual al de processadors disponibles. Cada fil executa una instància independent de l'algorisme amb una ciutat d'inici diferent. Finalment, es comparen totes les solucions obtingudes i es retorna la de menor cost. Aquesta és una forma de paral·lelisme de dades del tipus "embarrassingly parallel", ja que les tasques són completament independents i no requereixen sincronització, excepte en la recollida final de resultats.

C. Algorisme de Held-Karp (Programació Dinàmica)

L'algorisme de Held-Karp és un mètode exacte basat en programació dinàmica que resol el TSP de manera òptima. La seva complexitat, tot i ser exponencial, és significativament inferior a la de la força bruta ($O(n^2 \cdot 2^n)$ vs $O(n!)$), la qual cosa el fa factible per a instàncies de mida mitjana (fins a $n \approx 20$).

L'algorisme es basa en la definició d'una subestructura òptima. Sigui $C(S, j)$ el cost del camí més curt que comença a la ciutat 0, visita totes les ciutats del conjunt $S \subseteq 1, \ldots, n-1$ i acaba a la ciutat $j \in S$. La relació de recurrència es defineix com:

$$C(S, j) = \min_{k \in S, k \neq j} C(S \setminus j, k) + d_{kj}$$

El cas base és $C(k, k) = d_{0k}$ per a $k \in 1, \ldots, n-1$. La solució final del TSP s'obté calculant $\min_{j \in 1,\ldots,n-1} C(1, \ldots, n-1, j) + d_{j0}$.

La implementació (HeldKarpTSP.java) utilitza bitmasks per representar els subconjunts de ciutats S, una tècnica altament eficient. Un enter de n bits pot representar qualsevol subconjunt, on el i-è bit a 1 significa que la ciutat i pertany al conjunt. Les solucions dels subproblemes s'emmagatzemen en una taula de memoització (un HashMap o ConcurrentHashMap en la versió paral·lela) per evitar recàlculs.

Algorithm 3 Algorisme de Held-Karp amb Bitmasks

Require: Matriu de distàncies D, nombre de ciutats n
Ensure: Solució òptima (C_{opt}, P_{opt})
`Memo[<mask, last_city>] := (cost, parent)`
for $k = 1$ to $n-1$ **do**
  $Memo[(1 \ll k, k)] \leftarrow (D[0][k], 0)$
end for
for $s = 2$ to $n-1$ **do** ▷ Mida del subconjunt
  for cada combinació de s ciutats de $1..n-1$ **do**
    $mask \leftarrow$ crear bitmask per a la combinació
    for cada ciutat j a la combinació **do**
      $prev_mask \leftarrow mask \oplus (1 \ll j)$
      $min_cost \leftarrow \infty$
      for cada ciutat k a la combinació, $k \neq j$ **do**
        $cost \leftarrow Memo[(prev_mask, k)].cost + D[k][j]$
        if $cost < min_cost$ **then**
          $min_cost \leftarrow cost$
          $parent \leftarrow k$
        end if
      end for
      $Memo[(mask, j)] \leftarrow (min_cost, parent)$
    end for
  end for
end for
$C_{opt} \leftarrow \min_{j=1}^{n-1} Memo[((1 \ll n) - 2, j)].cost + D[j][0]$
Reconstruir camí P_{opt} a partir de $Memo$
return (C_{opt}, P_{opt})

a: Anàlisi de Complexitat:

- **Complexitat Temporal:** Hi ha $\binom{n-1}{s}$ subconjunts de mida s. Per a cada subconjunt, es consideren s possibles ciutats finals. Per a cadascuna, es fan $s-1$ operacions. La complexitat total és $\sum_{s=2}^{n-1} \binom{n-1}{s} \cdot s \cdot (s-1)$, que es pot demostrar que és $\Theta(n^2 \cdot 2^n)$.
- **Complexitat Espacial:** La taula de memoització emmagatzema una entrada per a gairebé cada parell (sub-

conjunt, ciutat final). El nombre d'entrades és aproximadament $(n-1) \cdot 2^{n-2}$, la qual cosa resulta en una complexitat espacial de $\Theta(n \cdot 2^n)$.

b: Paral·lelisme:

La classe HeldKarpTSP.java conté una implementació paral·lela, ParallelHeldKarpSolver, dissenyada per a $n \geq 12$. Aquesta versió utilitza un ForkJoinPool per paral·lelitzar el bucle principal que itera sobre les mides dels subconjunts. Dins del bucle, les combinacions de ciutats per a una mida de subconjunt donada es processen en paral·lel mitjançant un parallelStream(). Cada fil de treball calcula els costos per a un subconjunt de les combinacions i actualitza una taula de memoització concurrent (ConcurrentHashMap). Aquest disseny aprofita el paral·lelisme a nivell de dades, distribuint el càlcul dels estats de la programació dinàmica entre els nuclis disponibles.

D. Algorisme de Ramificació i Poda (Branch and Bound)

L'algorisme de Ramificació i Poda (B&B) és un altre mètode exacte per al TSP que explora l'espai de solucions de manera intel·ligent. Construeix un arbre de cerca on cada node representa un camí parcial. La seva eficàcia rau en la capacitat de "podar"branques senceres de l'arbre que no poden contenir una solució millor que la millor solució ja trobada.

Algorithm 4 Algorisme Branch and Bound amb Reducció de Matrius

Require: Matriu de costos D
Ensure: Solució òptima (C_{opt}, P_{opt})
$C_{opt} \leftarrow$ cost d'una solució inicial (ex: Greedy)
$Q \leftarrow$ cua de prioritat de nodes
$arrel \leftarrow$ node inicial (ciutat 0, nivell 1)
$arrel.cost \leftarrow$ calcular cota per reducció de D
$Q.afegir(arrel)$
while Q no és buida **do**
  $actual \leftarrow Q.extreure_min()$
  if $actual.cost \geq C_{opt}$ **then continue**
  end if ▷ Poda per cota
  if $actual$ representa un camí complet **then**
    $C_{opt} \leftarrow actual.cost$
    $P_{opt} \leftarrow actual.path$ **continue**
  end if
  for cada ciutat no visitada j **do**
    $fill \leftarrow$ crear node fill per a la ciutat j
    $fill.cost \leftarrow actual.cost + D[actual.vertex][j] +$ reducció_addicional
    if $fill.cost < C_{opt}$ **then**
      $Q.afegir(fill)$
    end if
  end for
end while
return (C_{opt}, P_{opt})

1) Algorisme Branch and Bound Seqüencial

La implementació BranchAndBound.java utilitza una cua de prioritat per gestionar els nodes de l'arbre de cerca, seguint una estratègia de "millor primer"(best-first search). Un node representa un camí parcial, i la seva prioritat ve donada per una cota inferior del cost de qualsevol solució completa que es pugui derivar d'aquest camí.

La clau de l'algorisme és la funció de cota inferior. Aquesta implementació utilitza el mètode de **reducció de matrius**. Per a un node donat, la cota inferior es calcula sumant el cost del camí parcial actual i el cost de reducció de la matriu de costos. La reducció consisteix a restar el valor mínim de cada fila i columna no cobertes pel camí parcial. Aquest cost de reducció representa el cost mínim addicional inevitable per completar el cicle.

A cada pas, l'algorisme extreu el node amb la cota inferior més baixa de la cua. Si la seva cota ja és superior a la millor solució coneguda (la cota superior), el node es descarta (poda). Altrament, s'expandeix generant nodes fills per a cada possible ciutat següent. Si un camí complet es genera i el seu cost és millor que la cota superior actual, aquesta s'actualitza.

a: Anàlisi de Complexitat:

- **Complexitat Temporal:** En el pitjor dels casos, l'algorisme ha d'explorar tot l'arbre, resultant en una complexitat $O(n!)$. Tanmateix, a la pràctica, l'eficàcia de la poda redueix dràsticament l'espai de cerca, fent-lo competitiu amb Held-Karp per a mides similars.
- **Complexitat Espacial:** La memòria depèn del nombre màxim de nodes emmagatzemats a la cua de prioritat, que pot ser exponencial en el pitjor cas.

2) Implementació Concurrent d'Alt Rendiment

La classe ConcurrentBranchAndBound.java representa una reimplementació de B&B que intenta optimitzar de manera paral·lela, dissenyada per a explortar el multi-nucli. Utilitza un ForkJoinPool per gestionar el paral·lelisme.

a: Optimitzacions i Estratègia de Paral·lelisme:

1) **Gestió d'Estat Global Atòmica:** La millor solució global (cost i camí) s'emmagatzema en variables atòmiques (`AtomicReference<Double>` i `AtomicReference<int[]>`). Això permet als fils actualitzar la millor cota superior de manera segura i eficient, sense bloquejos costosos, la qual cosa accelera la convergència de la poda a tot l'arbre.
2) **Cota Inferior Ràpida Precalculada:** En lloc de la costosa reducció de matrius a cada node, aquesta implementació precalcula les dues arestes de menor cost per a cada ciutat abans de començar la cerca. La cota inferior per a un subproblema es calcula llavors

en temps $O(n)$ (o més ràpid amb optimitzacions), sumant la meitat de la suma d'aquestes dues arestes mínimes per a cada ciutat no visitada. Això redueix dràsticament el cost de càlcul de la cota.

3) **Tasques Fork/Join Recursives:** La cerca es divideix en tasques (BranchAndBoundTask que hereta de RecursiveAction). Una tasca "paraigua"(UmbrellaBranchAndBoundTask) inicia el procés creant tasques per als primers nivells de l'arbre. Cada tasca explora un subarbre.
4) **Divisió de Treball i Llindar de Seqüenciació:** Una tasca només es divideix en subtasques paral·leles (forkSubtasks) si el nombre de branques a explorar és prou gran i la profunditat a l'arbre és petita. Això evita la sobrecàrrega de crear tasques per a subproblemes trivials, que s'executen de manera seqüencial dins del mateix fil.
5) **Reducció de Contenció i Optimització de Memòria:** S'utilitzen comptadors locals a cada fil per a les estadístiques, que només s'agreguen globalment de manera periòdica. A més, els objectes de camí i visitats es clonen només en crear una nova tasca, però es reutilitzen durant l'exploració recursiva seqüencial per minimitzar l'assignació de memòria. L'ús de bitmasks (visitedMask) per al seguiment de ciutats visitades permet comprovacions en O(1).

Aquesta implementació concurrent transforma el problema de cerca seqüencial en un paradigma de divideix i venceràs que hauria de ser més eficient, degut a que múltiples fils exploren diferents parts de l'arbre de cerca de forma coordinada i amb mínima contenció, aconseguint una acceleració significativa en màquines amb múltiples processadors.

V. Conclusió

En aquest treball s'ha aconseguit el disseny i la implementació d'una aplicació d'escriptori, basada en l'arquitectura Model-Vista-Controlador (MVC), per a la resolució i visualització del Problema del Viatjant de Comerç (TSP). S'han implementat i avaluat un ampli espectre d'algorismes, abastant des de l'enfocament exhaustiu de Força Bruta i l'heurística voraç del Veí Més Proper, fins a mètodes exactes d'alta complexitat com la programació dinàmica de Held-Karp i diverses variants de Ramificació i Poda (Branch and Bound). L'arquitectura del sistema, desenvolupada en Java 23, es fonamenta en patrons de disseny com el Factory Method per a la instanciació d'algorismes i un servei de notificacions desacoblat, garantint una estructura modular, extensible i mantenible.

Els resultats analítics i tècnics obtinguts validen l'abisme computacional existent entre les diferents estratègies.La solució de Força Bruta, amb la seva complexitat factorial $O(n!)$, esdevé intractable ràpidament, sent només pràctica per a instàncies trivials ($n \leq 10$), mentre que l'heurística Greedy, de cost $O(n^2)$, ofereix solucions ràpides però no òptimes. En canvi, els mètodes exactes avançats demostren una eficiència superior. L'algorisme de Held-Karp, amb una complexitat de $O(n^2 \cdot 2^n)$, redueix dràsticament l'espai de cerca respecte a la força bruta i permet resoldre de manera òptima instàncies de mida mitjana. Paral·lelament, la tècnica de Ramificació i Poda, malgrat la seva complexitat de pitjor cas també exponencial, es revela com un mètode altament competitiu gràcies a l'eficàcia de les seves funcions de poda, especialment en la seva versió avançada que utilitza reducció de matrius per ajustar les cotes inferiors.

A més, un dels eixos centrals del projecte ha estat l'aplicació de tècniques de concurrència avançada per mitigar la complexitat inherent del TSP. S'han desenvolupat versions paral·lelitzades dels algorismes clau, aprofitant el framework Fork/Join de Java. La implementació paral·lela de Held-Karp distribueix el càlcul dels estats de la programació dinàmica entre els nuclis disponibles mitjançant 'parallel streams'.

Les mesures empíriques realitzades indiquen que, malgrat l'estratègia teòrica de paral·lelisme, la implementació concurrent de Ramificació i Poda resulta més lenta que la seva versió seqüencial en la pràctica. Això s'explica principalment pels següents factors:

Contenció i overhead atòmic: L'ús de 'AtomicReference<Double>' i altres variables atòmiques per gestionar de forma segura la millor cota global introdueix cost addicional en cada actualització i lectura, ja que aquestes operacions impliquen instruccions de sincronització a nivell de maquinari i possibles barreres de memòria.

Creació i planificació de tasques: El model Fork/Join genera subtasques recursives ('RecursiveAction') fins i tot per subproblemes petits, amb el consegüent cost de creació, encolament i despatx de cada tasca[cite: 169, 172]. Aquest overhead de gestió de tasques supera el benefici de paral·lelització en instàncies de mida moderada.

Granularitat insuficient: La divisió del treball no sempre és prou àmplia per amortir el cost de les operacions atòmiques i de planificació. En ramificacions de profunditat intermèdia, el nombre de branques és reduït i el cost de sincronització domina el temps total.

Per tant, en instàncies amb n fins a 15–20 ciutats, la versió seqüencial de Branch and Bound, que fa servir reducció de matriu sense accés atòmic global i processa de manera local cada subarbre, demostra millor eficiència. En futurs treballs es podrien explorar estratègies de tasques amb llindars més alts —evitant paral·lelitzar subproblemes petits— i estructures de dades no atòmiques (per exemple, buffers per lots d'actualitzacions) per minimitzar la contenció i recuperar el benefici del paral·lelisme.

En resum, les solucions desenvolupades no només implementen els algorismes canònics per al TSP, sinó que exploren i demostren l'impacte crucial de les optimitzacions algorísmiques i l'exploració del paral·lelisme d'alt rendiment. Els mètodes exactes avançats, i en particular les implementacions seqüencials optimitzades, superen de manera clara les limitacions de les aproximacions més simples i fan factible la resolució òptima d'instàncies complexes del problema.

L'arquitectura resultant, que combina un disseny de programari robust amb tècniques algorísmiques sofisticades, valida l'enfocament emprat i constitueix una eina potent i escalable per a l'anàlisi del Problema del Viatjant de Comerç.

Referències

[1] Oracle, *Fork/Join Framework*, Java Tutorials, 2014.
[2] S. Klymenko, *Java ForkJoinPool: A Comprehensive Guide*, Medium, 2021.
[3] A. Dix et al., *Model-View-Controller (MVC) and Observer*, Univ. of North Carolina, 2016.
[4] Sommerville, I., *Software Engineering*, 10th ed., Addison-Wesley, 2015.
[5] Chacon, S. and Straub, B., *Pro Git*, Apress, 2014.
[6] Cockburn, A., *Agile Software Development: The Cooperative Game*, Addison-Wesley, 2006.
[7] Loeliger, J. and McCullough, M., *Version Control with Git*, O'Reilly Media, 2012.
[8] Sutherland, J., *Scrum: The Art of Doing Twice the Work in Half the Time*, Crown Business, 2014.
[9] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
[10] Reenskaug, T., *Models, Views and Controllers*, 1979.
[11] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M., *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, Wiley, 1996.
[12] Dix, A., Finlay, J., Abowd, G. and Beale, R., *Human-Computer Interaction*, Prentice Hall, 2004.
[13] Fowler, M., *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.
[14] Joshua Bloch, *Creating and Destroying Java Objects*, Drdobbs, 2008
[15] Goetz, B. et al., *Java Concurrency in Practice*, Addison-Wesley, 2006.
[16] Lea, D., *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, 2000.
[17] Amdahl, G. M., *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*, AFIPS Conference Proceedings, 1967.
[18] Herlihy, M. and Shavit, N., *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2008.
[19] Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C., *Introduction to Algorithms*, 3rd ed., MIT Press, 2009.
[20] Little, J. D. C., Murty, K. G., Sweeney, D. W., and Karel, C., *An Algorithm for the Traveling Salesman Problem*, Operations Research, 11(6), 972-989, 1963. https://doi.org/10.1287/opre.11.6.972
[21] Held, M. and Karp, R. M., *A dynamic programming approach to sequencing problems*, Journal of the Society for Industrial and Applied Mathematics, 10(1), 196-210, 1962. https://doi.org/10.1137/0110015
[22] Rosenkrantz, D. J., Stearns, R. E., and Lewis, P. M. II, *An Analysis of Several Heuristics for the Traveling Salesman Problem*, SIAM Journal on Computing, 6(3), 563-581, 1977. https://doi.org/10.1137/0206041
[23] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C., *Introduction to Algorithms, 3rd ed.*, MIT Press, 2009.
[24] Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C., *Introduction to Algorithms*, 3rd ed., MIT Press, 2009.
[25] Kleinberg, J. and Tardos, É., *Algorithm Design*, Pearson, 2006.
[26] Aho, A. V., Hopcroft, J. E. and Ullman, J. D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
[27] Sipser, M., *Introduction to the Theory of Computation*, 3rd ed., Cengage Learning, 2012.
[28] Tarjan, R. E., *Data Structures and Network Algorithms*, SIAM, 1983.

Annex:

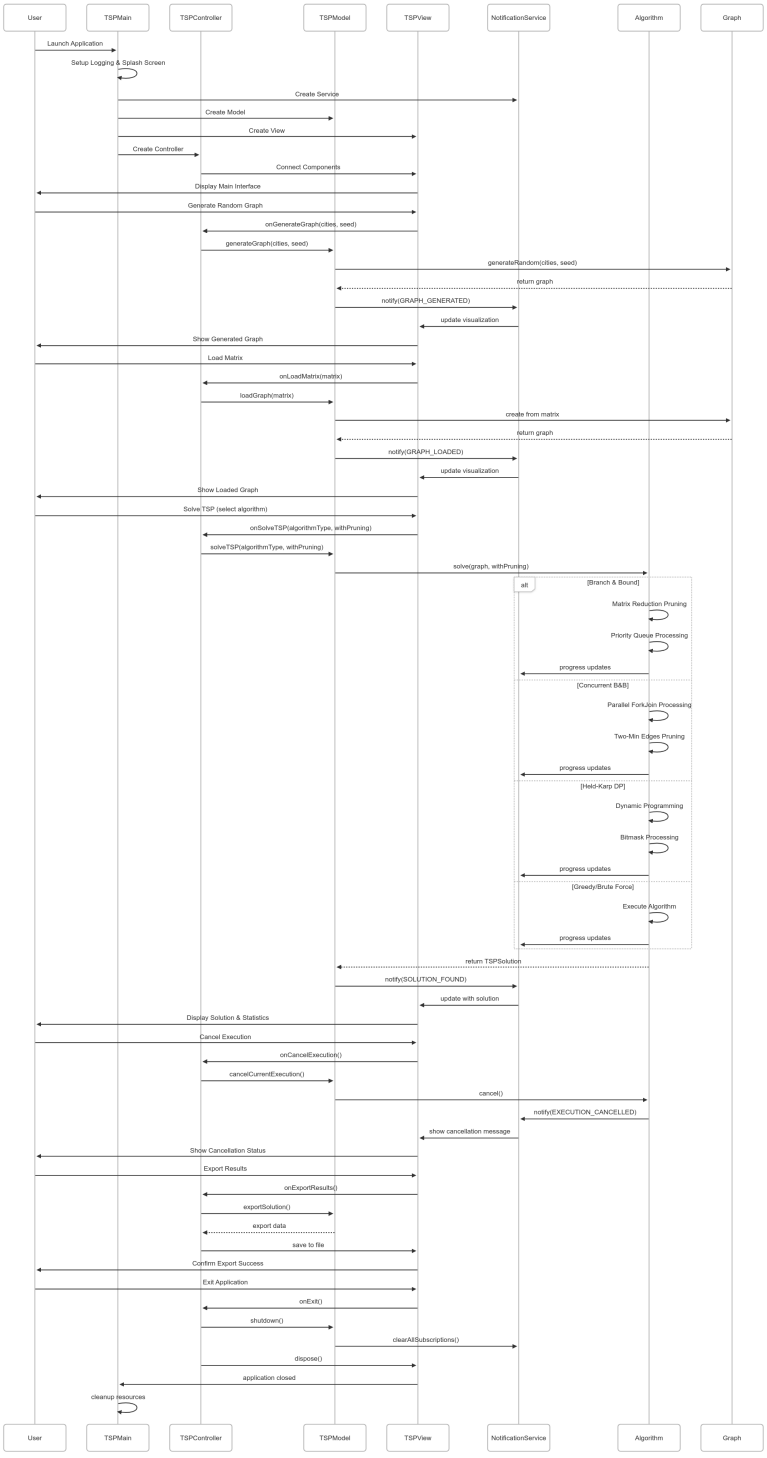

Figura 4. Diagrama de Fluxe