Marc Provinciale Isach

Computació d'algoritmes genètics en sistemes Serverless

TREBALL DE FI DE GRAU

dirigit pel Dr. Marc Sánchez Artigas

Grau d'Enginyeria Informàtica

UNIVERSITAT ROVIRA I VIRGILI

Tarragona

2021

Resum.

La gran quantitat d'informació que circula actualment fa que es necessitin sistemes cada vegada més potents, que si s'haguessin de crear, administrar i mantenir per usuaris particulars, no serien rendibles, però gràcies a les solucions Cloud podem tractar aquesta informació.

En aquest treball s'ha estudiat el comportament dels algoritmes genètics resolent el problema del Viatjant de Comerç o Travelling Salesman Problem quan es fa la seva execució en una màquina local i utilitzant la creixent tecnologia Cloud anomenada Serverless Computing, utilitzant el framework Lithops-Cloud de Python per fer la interacció amb el proveïdor Cloud. D'aquesta forma, s'han analitzat els resultats obtinguts en els dos casos i els beneficis que pot aportar la tecnologia Serverless en el tractament de dades massives tant per empreses com per a investigadors, com poden ser la millor eficiència de recursos, la disminució de costos i la facilitat d'utilització. Tot i això, també té alguns inconvenients com poden ser l'inici en fred o les inestabilitats en els temps d'execució, però tot això es veu durant l'obtenció i l'anàlisi de resultats.

Resumen.

La gran cantidad de información que circula actualmente hace que se necesiten sistemas cada vez más potentes, que si se tuviesen que crear, administrar y mantener por usuarios particulares no seria rentables, pero gracias a soluciones Cloud se puede tratar esta información.

En este proyecto se ha estudiado el comportamiento de los algoritmos genéticos resolviendo el problema del Viajante de Comercio o Travelling Salesman Problem cuando se hace su ejecución en una máquina local y utilizando la creciente tecnología Cloud llamada Serverless Computing, utilizando el framework Lithops-Cloud para hacer la interacción con el proveedor Cloud. De esta forma, se han analizado los resultados obtenidos en ambos casos y los beneficios que aporta la tecnología Serverless en el tratamiento de datos masivos tanto para empresas como para investigadores, como pueden ser la mejora de la eficiencia de recursos, la disminución de costes y la facilidad de utilización. Aun así, también tiene algunos inconvenientes como puede ser el inicio en frio o las inestabilidades en los tiempos de ejecución, pero todo esto se ve durante la obtención y análisis de resultados.

Abstract.

The large amount of information in circulation nowadays means that more powerful systems are needed, which if they had to be created, administrated and maintained by particular users, they would not be cost-effective, but thanks to Cloud solutions, we can process this information.

In this project it has been studied the behaviour of genetic algorithm solving the Travelling Salesman Problem when they are executed in a local machine and using the increasing Cloud technology named Serverless Computing, using Lithops-Cloud framework to make the interaction with the Cloud provider. In this way, the results obtained in both cases have been analysed, and the benefits that give Serverless technology in the processing of massive data for both companies and researchers, such as the increase resource efficiency, the diminution of costs and the ease of use. Even though, it also has some drawbacks, such as cold starts or instabilities in the execution times, but all this can be seen during the obtention and analysing of results.

Índex

Índex de taules

Índex de figures

Índex de pseudocodis

1 Introducció

Actualment la quantitat de dades que es generen, estudien, analitzen, modifiquen, destrueixen i guarden són molt importants per molts aspectes de la societat actual per la quantitat d'informació que proporcionen. Moltes companyies es dediquen a fer estudis sobre que compraran els seus clients, les preferències que tenen o per fer prediccions científiques com poden ser el canvi climàtic o l'expansió de l'univers. En tots aquests aspectes el tractament de les dades és molt important, però la mida d'aquestes dades cada vegada és més gran al recopilar cada vegada més informació que potser per molts és menyspreable, però que per a les companyies, desenvolupadors o estudis científics poden aportar molta informació.

Per tractar totes aquestes dades es necessiten sistemes cada vegada més potents, que si tracten amb quantitats petites potser amb un ordinador és suficient, però per la quantitat de dades que tracten les empreses i científics no és suficient. Per aquest motiu han començat a sorgir les tecnologies Cloud les quals proporcionen diferents serveis a través d'Internet els quals es contracten segons les necessitats. Aquests serveis poden ser de Software per utilitzar programes o aplicacions online; de Plataformes per a desenvolupar aplicacions; o d'Infraestructures per a què l'empresa administri la plataforma i les aplicacions i només contracti els recursos hardware necessaris.

En aquest cas, aquest treball de final de grau es centra a analitzar una nova tecnologia dintre del món Cloud, la tecnologia Serverless.

1.1 Descripció del projecte

Aquest treball de final de grau es centra a analitzar els beneficis que pot proporcionar la tecnologia **Serverless** i en quins casos pot ser ideal utilitzar-la. Per a fer-ho s'ha estudiat com es comporten els **Algoritmes Genètics** resolent el problema del **Travelling Salesman Problem (TSP) o Problema del Viatjant de Comerç** i estudiar els beneficis que pot aportar la tecnologia Serverless en executar algoritmes genètics i comparar-los a l'executar en una màquina local.

Durant el treball es fa una explicació de tots els conceptes previs que es tracten, per a després entendre de forma més senzilla la implementació realitzada i els resultats obtinguts. Després, per facilitar la utilització de la tecnologia Serverless, s'ha utilitzat el projecte Lithops Cloud de Python i s'ha analitzat els diferents components de la tecnologia que utilitza, el seu rol i el flux d'execució que té per entendre millor la tecnologia. Per últim, es fa una explicació de la implementació realitzada i s'analitzen els resultats obtinguts pels diferents casos proposats.

1.2 Objectius del projecte

L'objectiu principal d'aquest treball és el de **mostrar els beneficis** de les cada vegada més conegudes tecnologies **Cloud**, concretament centrant-se en la tecnologia **Serverless**.

A més a més, altres objectius han sigut el de proposar una possible implementació amb algoritmes genètics resolent el problema del TSP i analitzant el seu comportament de forma exhaustiva tant de forma local com amb un sistema Serverless; o també el de familiaritzar-se amb les diferents tecnologies Cloud actuals i els seus diferents proveïdors.

2 Recerca prèvia

Per a començar amb l'elaboració del projecte al principi es van haver de decidir quines tecnologies utilitzar i fer una recerca de les bases més importants del projecte com són els algoritmes genètics, el Travelling Salesman Problem o els sistemes Serverless. En aquest punt s'explica les tecnologies utilitzades i els conceptes més importants en profunditat, per a després poder comprendre més fàcilment les decisions de la implementació.

2.1 Tecnologies utilitzades

El llenguatge de programació utilitzat ha sigut **Python**, que va ser escollit degut a la gran quantitat de llibreries que té i sobretot per la seva llibreria **lithops-Cloud**, de la qual es parlarà més endavant, però és la part fonamental per a la interacció amb el núvol.

Python és un llenguatge de programació d'escriptura dinàmica [1]d'alt nivell que es basa en la llegibilitat, el qual pot servir tant per la Programació Orientada a Objectes[2] com per a la Programació Funcional[3]. És utilitzat per tot tipus de tasques degut a la seva facilitat d'aprenentatge, quantitat de recursos lliures de cost i tot el seu potencial, però sobretot s'utilitza en entorns d'anàlisi de dades per la quantitat de llibreries completes i ben documentades que hi ha per aquest àmbit.

Per a poder tenir una còpia disponible del codi en qualsevol lloc i moment i tenir un històric dels diferents canvis que ha anat seguint el projecte s'ha creat un repositori **Git** [4], on s'anaven especificant els diferents canvis que es realitzaven al projecte per a poder recuperar qualsevol estat del projecte en qualsevol moment i tenir una còpia d'aquest disponible des de qualsevol lloc.

El problema escollit ha sigut el del **Travelling Salesman Problem (TSP)** degut a la seva importància en molts dels àmbits actuals i la seva idoneïtat a l'hora d'utilitzar algoritmes genètics. Pot servir tant com per trobar la ruta òptima per viatjar per un conjunt de ciutats especificades, trobar el camí òptim per a la distribució d'una xarxa elèctrica o logística o com unir els punts de soldadura d'una placa base per a fer les connexions de la forma més idònia, entre altres moltes més aplicacions.

Per a poder resoldre el problema del TSP s'ha implementat un **algoritme genètic**, el qual es basa en la idea dels algoritmes evolutius[5]. S'ha decidit utilitzar-lo per la seva capacitat de reduir el temps de cerca d'una solució òptima pel TSP, a més de veure com la seva execució al núvol ajuda a trobar solucions òptimes més fàcilment.

Per últim, dintre de les diferents tecnologies de les quals disposa el núvol s'ha decidit utilitzar la computació Serverless, per la seva simplicitat en aplicacions on no s'ha de guardar els recursos i els temps de computació són curts, és a dir, per a l'execució de programes d'iniciar, fer una certa tasca i finalitzar. A més, per guardar possibles recursos que han

[1] L'escriptura dinàmica consisteix a verificar el codi durant el temps d'execució en comptes de durant la compilació del projecte.

[2] La Programació Orientada a Objectes es basa en el concepte de programar amb objectes, els quals tenen uns atributs que el caracteritzen i uns mètodes que són les accions o processos que realitza.

[3] La Programació Funcional es basa en el concepte de programar seguint una estructura principal de funcions per realitzar les diferents accions que volem fer.

[4] Software per a realitzar un seguiment dels canvis que té un projecte durant el seu desenvolupament, d'una forma ordenada i coordinada entre els diferents integrants.

[5] Els algoritmes evolutius són mètodes d'optimització i cerca de solucions basats en l'evolució biològica.

d'utilitzar totes les execucions al núvol com pot ser el mapa del TSP, s'ha decidit utilitzar un emmagatzematge d'objectes al núvol en calent.

2.2 Travelling Salesman Problem (TSP)

El **Travelling Salesman Problem** o TSP [1][2] és un problema **d'optimització combinatòria[6] NP-Hard**[7] el qual vol respondre a la pregunta de "Quin és el camí més curt possible que visita exactament un cop cada ciutat i finalitza retornant a la ciutat d'origen?"

Va ser formulat el 1930 i és un dels problemes d'optimització més estudiats. Tot i que és computacionalment complex es coneixen heurístiques i mètodes per resoldre el problema des de cent fins a milers de ciutats.

S'utilitza en moltes aplicacions, però la versió més simple serveix per a la planificació, la logística i la fabricació de circuits electrònics. Modificant-lo es pot utilitzar per a la seqüenciació de l'ADN.

En el pitjor dels casos és probable que el temps d'execució per qualsevol algoritme que resolgui el TSP el temps augmenti de forma exponencial respecte el nombre de ciutats.

Per a resoldre el problema es representa N! solucions possibles, tot i que si no ens importa l'inici el nombre de solucions a examinar es redueix a (N-1)!. Si no importa la direcció de desplaçament el nombre de solucions es redueix a la meitat, és a dir, només s'ha de considerar **(N-1)!/2 solucions possibles**.

Per a resoldre problemes amb poques ciutats el problema no té molta complexitat, per exemple per a 10 ciutats necessitem (10-1)!/2 = 181.440, però al ser factorialment creixent amb 30 ciutats necessitem (30-1)!/2 = 4,420 * 10^{30}, el que fa que la potencia necessària per a resoldre'l sigui molt gran. Si intentem resoldre el problema provant totes les permutacions possibles i veure quina ha sigut la millor solució el temps d'execució seria d'un factor polinòmic d'ordre ***O*(n!)**.

El problema es pot representar a través d'un **graf ponderat no dirigit**[8], on els vèrtexs corresponen a les ciutats a visitar, els camins són les arestes i la distància dels camins és el pes de les arestes. Hi ha també la versió de graf ponderat dirigit, on pot haver-hi diferents pesos en cada direcció del camí o pot representar que només una de les direccions existeix. Els grafs ponderats no dirigits també se'ls anomena TSP simètrics i als dirigits TSP asimètrics.

Algunes de les restriccions que té aquest problema són que només es pot visitar cada ciutat un cop, s'ha d'acabar retornant a la ciutat d'origen i depenent de si ens interessa o no la ciutat d'origen estarà especificada o podrà ser qualsevol.

[6] L'optimització combinatòria és una branca de l'optimització en matemàtiques on es resolen problemes que es creuen difícilment generals, explorant tot l'espai de solucions que acostuma a ser molt gran. Redueixen l'espai de cerca buscant de forma eficient.

[7] En la teoria de la complexitat computacional, els problemes NP-Hard són el conjunt de problemes de decisió que contenen els problemes H tals que tot problema L en NP poden ser transformats en H. Aquests problemes com a mínim seran tan difícils com un problema de NP.

[8] Un graf ponderat no dirigit és la representació de les connexions de diferents punts, on cada connexió té un cost de viatjar des d'un punt X a un punt Y (ponderat) i no importa si el viatge és del punt X al punt Y o a la inversa, que tindrà el mateix cost (no dirigit).

2.3 Algoritmes genètics

Els algoritmes genètics (AG) [3][4] s'anomenen així per inspirar-se en l'evolució biològica i la seva base genètica-molecular. Aquests algoritmes pretenen evolucionar una població d'individus seguint el procés aleatori de **l'evolució biològica** i amb algun tipus de selecció d'individus per a seguir les regles de la **selecció natural**, on **els individus més adaptats sobreviuen** i els menys adaptats acaben desapareixent.

El seu funcionament es basa a fer evolucionar una població d'individus, on cada individu representa el que s'anomena un **cromosoma**, que té codificada la informació de la solució que proposa aquell individu, on cada posició del cromosoma representa un gen de la solució. Aquests cromosomes poden ser codificats de diferents formes segons cada problema, però normalment es codifiquen de forma binària.

Per a tenir un criteri de quins individus són millors o pitjors solucions que la resta i poder-los comparar s'utilitza una funció per avaluar l'individu, anomenada **funció de fitness**.

Durant l'evolució de la població es passarà per unes certes fases durant un seguit de generacions, fins que la variació de la solució sigui mínima durant unes quantes generacions, el qual significarà que s'ha arribat a un màxim o mínim local[9] o fins i tot al màxim o mínim general; o per un límit de generacions que s'especificarà. Depenent de la implementació que es segueixi aquest criteri de finalització serà diferent en cada problema adaptant-lo a les necessitats que es tingui.

Els passos pels quals passa la població són primerament la **inicialització** de la població per a tenir uns individus variats amb els quals començar, després, fins que el criteri de finalització de l'evolució es compleixi, es **calcularà el fitness** de cada individu de la generació, es **seleccionarà** els diferents individus que crearan la nova població, es **creuaran** els individus seleccionats per a generar una nova població, s'aplicarà un criteri de **mutació** dels gens dels nous individus de forma aleatòria i s'afegiran els nous individus a la nova població per substituir l'antiga. Un cop el criteri de finalització es compleix es retorna el millor individu de l'última generació que s'ha utilitzat per a fer tot el procés, i aquest serà la solució que proposa l'algoritme.

2.3.1 Inicialització de la població

La població inicial per fer front al problema a resoldre amb l'AG ha de ser **prou gran per facilitar la diversitat inicial** de la població i les seves posteriors generacions, i dintre de l'espai de cerca del problema. D'aquesta forma es troba la solució general al problema de forma més senzilla.

També es poden "definir" alguns individus inicials per a facilitar l'obtenció de solucions òptimes cap a una àrea en específic, i així centrar l'espai de cerca en una zona més focalitzada.

[9] Un **màxim o mínim local** és un individu proposat per l'AG el qual pot ser que no sigui el millor individu que es podria obtenir en el problema, però que a causa del seu fitness i que s'ha convertit en l'individu majoritari en el problema els nous individus proposats resulten pitjors o molt semblants a l'individu, i que per aconseguir sortir s'hauria de produir un canvi d'individus considerable i molt poc probable.

2.3.2 Funció de fitness

La funció de fitness és la que **avalua com és de bo un individu com a solució** al problema proposat. Segons si el que es vol és disminuir o augmentar aquest fitness es parla d'un problema de minimització (disminuir) o maximització (augmentar).

Aquesta funció és **diferent per a cada problema**, i pot ser molt senzilla o molt complexa de calcular. Com la funció de fitness s'ha d'avaluar per a tots els individus durant moltes generacions, a vegades es redueixen problemes grans, on la funció de fitness seria molt complexa, a problemes més petits on la funció de fitness és més senzilla, per així reduir el temps de càlcul de la funció de fitness i el temps general en executar l'algoritme.

2.3.3 Selecció d'individus

Durant la selecció d'individus [5] es seleccionarà quins individus formaran part en la creació de la nova població de la següent generació.

Per a seleccionar individus existeixen una sèrie de mètodes, des de seleccionar els individus que millor fitness tenen en la població actual (elitisme) fins a fer una selecció de forma aleatòria sense importar el valor fitness. Si es seleccionen només els millors individus es pot trobar que es caigui molt sovint en un màxim o mínim local, o si es seleccionen de forma molt aleatòria es pot trobar que es necessiten moltes més generacions per a aconseguir trobar una solució òptima o propera, ja que l'evolució serà molt més lenta. La majoria de mètodes que s'utilitzen per a seleccionar individus **utilitzen el fitness** per ajudar a fer que els **millors individus** de la població tinguin **més probabilitats** de ser seleccionats.

Alguns mètodes poden ser "Roulette Wheel Selection", on els individus amb major fitness tenen un valor total de la ruleta més gran i per tant més probabilitat, "Rank Selection", que s'utilitza principalment quan la variació de fitness entre individus és molt lleugera, "Tournament Selection", "Elitism Selection", on els millors individus són seleccionats, entre altres.

El mètode que s'ha decidit utilitzar per a la selecció d'individus ha sigut el **Tournament Selection**. Consisteix a seleccionar X individus de forma aleatòria de la població, i es classifiquen segons el seu valor fitness. El que millor fitness té d'aquests X seleccionats, és seleccionat per a crear la següent generació. I això es repeteix fins a obtenir Y individus per a fer el creuament. D'aquesta forma es té aleatorietat per a deixar que individus amb menor fitness puguin ser seleccionats, i es facilita l'evolució al seleccionar els millors individus d'aquests grups aleatoris.

2.3.4 Creuament de seleccionats

Un cop s'ha seleccionat els individus per a crear la següent generació s'hauran de **creuar entre ells** [6] com si es reproduïssin i agafessin característiques de cada pare. Per a creuar els individus es poden anar seleccionant en parelles de dos, de forma aleatòria les parelles o seguint algun criteri, i formar dos fills de la parella seleccionada, i repetir el procés fins a tenir el mateix nombre d'individus que a la població inicial.

Per a creuar els individus depenent del problema serà més senzill o complicat pel fet que alguns problemes han de mantenir una sèrie de condicions en els seus individus, com pot ser no repetir gens, el gen inicial ha de ser sempre el mateix o alguna altra.

Alguns mètodes de creuament són el **"One-point crossover"**, on de forma aleatòria es selecciona un gen aleatori que farà de punt mitjà entre la informació del primer pare i la informació del segon, i el primer fill disposarà a la primera part de la informació del primer pare i a la segona part del segon pare, i el segon fill serà a l'inrevés. També existeix el **"Two-**

point crossover” i **“K-point crossover”**, els quals segueixen el mateix principi que el “One point crossover” però amb K punts de partició aleatoris. Per últim, existeixen els **creuaments de llistes ordenades**, on el problema TSP es troba, i que són **mètodes de creuament especialitzats** per a conservar les restriccions del problema que es poden tenir, com per exemple podria ser que dos parells de gens han d’anar sempre units o altres.

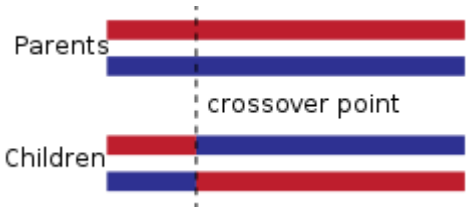


Figura 1. One-Point Crossover

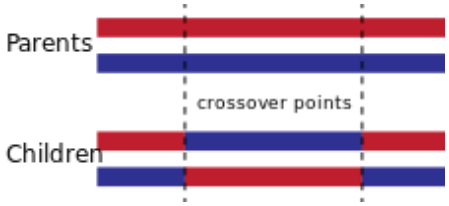


Figura 2. Two-Point Crossover

2.3.5 Mutació d’individus

Per a **mantenir la diversitat** de la població durant tota l’execució de l’algoritme s’utilitza la mutació [7] per variar algun gen de forma aleatòria i així trobar solucions les quals sense la mutació no haurien sigut possibles d’obtenir. D’aquesta forma es poden obtenir **individus més adaptats** gràcies a alguna mutació o també a l’inrevés, es poden obtenir individus pitjors per culpa d’una mutació. Tot aquest procés és **aleatori**, i per a mantenir una bona evolució dels individus s’ha de mantenir aquesta probabilitat de mutació baixa, per a poder conservar també la major part dels diferents individus i no tenir individus totalment aleatoris en cada generació. Tot això per a mantenir una mica de variació dels individus i poder sortir d’un màxim o mínim local.

Per a poder fer aquest procés de mutació hi ha diferents estratègies, però les més conegudes són l’intercanvi de posicions entres dos gens (mutació per permutació) o la negació d’un bit si s’està tractant amb cromosomes binaris (mutació binària).

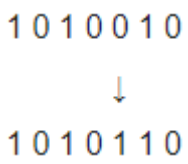


Figura 3. Mutació binària

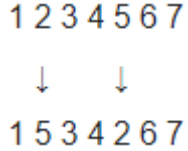


Figura 4. Mutació per permutació

2.4 Computació Serverless

La computació Serverless [8] és un **model d'execució al núvol** que aprovisiona de recursos **segons les necessitats** del consumidor, i treu tota la responsabilitat de les tasques comuns de gestió dels servidors al consumidor, ja que les realitza el proveïdor. Això permet que els consumidors només s'hagin de centrar en el desenvolupament de la seva aplicació i no en els recursos necessaris.

La computació Serverless permet, a més a més de **facilitar** la gestió i operació de la infraestructura dels recursos del software, executar el codi **segons la demanda** que el consumidor tingui, en contenidors "stateless"[10] i que escala de forma transparent[11] a l'usuari; i per últim, permet que l'usuari només pagui pels recursos només utilitzats, per tant redueix el cost de l'aplicació.

Com totes les tecnologies té els seus beneficis i els seves desavantatges. Com es pot veure en la taula següent:

Avantatges	Desavantatges
Simplifica l'execució d'aplicacions al núvol al no haver-se de preocupar de l'arquitectura del núvol. A més a més es pot programar en pràcticament qualsevol llenguatge que es vulgui.	Càrregues de **treball estable o previsibles no ofereixen un estalvi considerable** i acaben sent més simples i econòmiques tecnologies Cloud més tradicionals.
Només **es paga pels recursos utilitzats**, ja que els preus van segons les execucions que es realitzen.	**L'inici en fred** pot ser un problema a l'iniciar l'aplicació, ja que molts cops els servidors s'inicien quan es fa la petició i tarden un temps a escalar de forma idònia.
En algunes càrregues de treball, sobretot en tasques on la paral·lelització és important, **l'execució pot ser més ràpida i més eficient** segons el cost que altres alternatives.	La **monitorització i la depuració** es fan més complexes a l'hora de treballar en sistemes distribuïts, i es magnifica més en aquest entorn.
Pràcticament tota la **informació** generada pel servidor sobre el sistema i els seus usuaris és **visible**.	L'arquitectura Serverless està pensada per **aprofitar tot l'ecosistema del núvol** i no només una sola aplicació, per això molts cops les companyies no acaben adaptant-lo al no acabar-se de fiar.

Taula 1. Avantatges i desavantatges de la computació Serverless

[10] Un contenidor stateless es tracta d'un lloc d'execució per a una aplicació que no llegeix ni guarda la informació sobre l'estat de l'aplicació d'execucions anteriors o per a execucions posteriors, és a dir, tota la informació que necessita per executar-se la llegeix o la guarda durant la seva execució en aquest contenidor i no dependrà d'altres execucions.

[11] Escalar de forma transparent significa que tot el procés de reserva de més o menys recursos segons la càrrega de treball actual es farà de forma automàtica sense que l'usuari se'n adoni del procés d'escalat.

Dintre la computació Serverless hi ha diferents serveis per a poder utilitzar segons el que més interessi, però per aquest treball interessen principalment dos d'aquests: **el Function-as-a-Service** i **l'Object Storage**.

El **Function-as-a-Service o FaaS [9]** és principalment el servei central de la tecnologia Serverless. Representa el motor de computació principal del sistema Serverless, és a dir, permet executar el codi de l'aplicació sense preocupar-se dels recursos necessaris.

L'Object Storage [10] és el servei principal per emmagatzemar grans quantitats de dades o que no tenen una estructura definida per a guardar les dades en bases de dades tradicionals. Els objectes són unitats de dades que s'emmagatzemen en un entorn estructuralment pla. No existeixen carpetes, directoris o estructures jeràrquiques complexes com en els sistemes de fitxers tradicionals. Cada objecte és simple, són repositoris auto-continguts que inclouen les dades, metadades i un identificador únic per a poder treballar amb ell. Per tant, elimina la complexitat i els problemes d'escalabilitat dels sistemes de fitxers jeràrquics amb carpetes i directoris.

2.5 Lithops-Cloud

Lithops [11] és un framework[12] per a Python per a **l'anàlisi de grans quantitats de dades** o **treballs en paral·lel massius**, el qual proporciona una API universal per **crear aplicacions paral·leles al núvol**.

Permet executar el codi de l'aplicació sense modificacions de forma massiva a les principals plataformes de computació Serverless. Extreu la responsabilitat dels coneixements que es necessiten i com es desplega i executa l'aplicació en servidors Serverless, per tant encara facilita més l'execució de les aplicacions al núvol, ja que permet executar en tots els proveïdors principals i sense haver de tenir molts coneixements de les tecnologies del núvol. Està pensat principalment per entorns paral·lels amb poca o nul·la comunicació entre els processos, però també suporta la comunicació entre processos.

Per a poder-lo utilitzar simplement es necessita crear un compta Cloud a algun d'aquests proveïdors:

Figura 5. Proveïdors Cloud on executar Lithops.

Seguidament s'haurà de configurar un arxiu de configuració de Lithops especificant el proveïdor Cloud que s'utilitza i les credencials pròpies per a executar els serveis FaaS i Object Storage.

Finalment, s'haurà d'afegir en el codi de l'aplicació unes poques línies per a fer la paral·lelització.

[12] Un framework és un software ja creat que proporciona una forma estàndard de construir aplicacions i és universal, reutilitzable i facilita el desenvolupament al donar funcionalitats particulars que poden servir pel desenvolupament d'una aplicació.

Lithops disposa de dos APIs de computació (Futures API i Multiprocessing API) i dos APIs d'emmagatzematge (Storage API i Storage OS API).

La Futures API permet executar el codi directament al núvol i la Multiprocessing API permet utilitzar la llibreria de multiprocessament de Python per a executar funcions de forma paral·lela en un entorn Cloud.

La Storage API permet crear un emmagatzematge al núvol sense preocupar-se dels detalls d'implementació i la Storage OS API permet interactuar de forma transparent amb l'emmagatzematge al núvol.

A més a més, Lithops disposa de tres modes diferents d'execució:

- El **Localhost Mode** permet executar les funcions a la nostra màquina utilitzant processos. Com si es fes una simulació de l'execució al núvol.
- El **Serverless Mode** permet executar les funcions utilitzant els serveis Serverless que s'han configurat. Cada invocació correspon a una tasca paral·lela al Cloud en un entorn aïllat, anomenada Worker.
- El **Standalone Mode** permet executar funcions utilitzant una o múltiples màquines virtuals al núvol. Cada màquina virtual executa les funcions utilitzant processos paral·lels com en el Localhost mode.

Per a veure com a exemple el temps que tarda a realitzar cada activació, finalització o com es distribueixen aquestes activacions en el temps hi ha les següents figures:

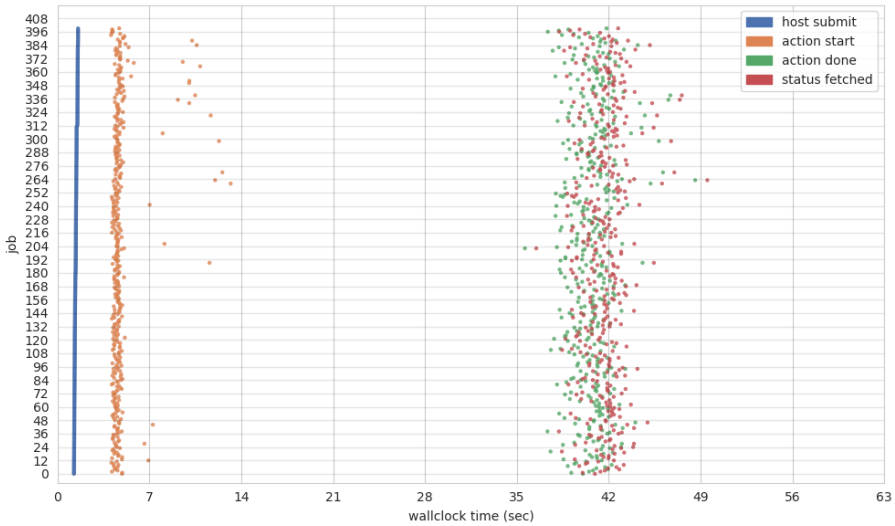


Figura 6. Exemple 1 de l'activació de funcions amb Lithops-cloud

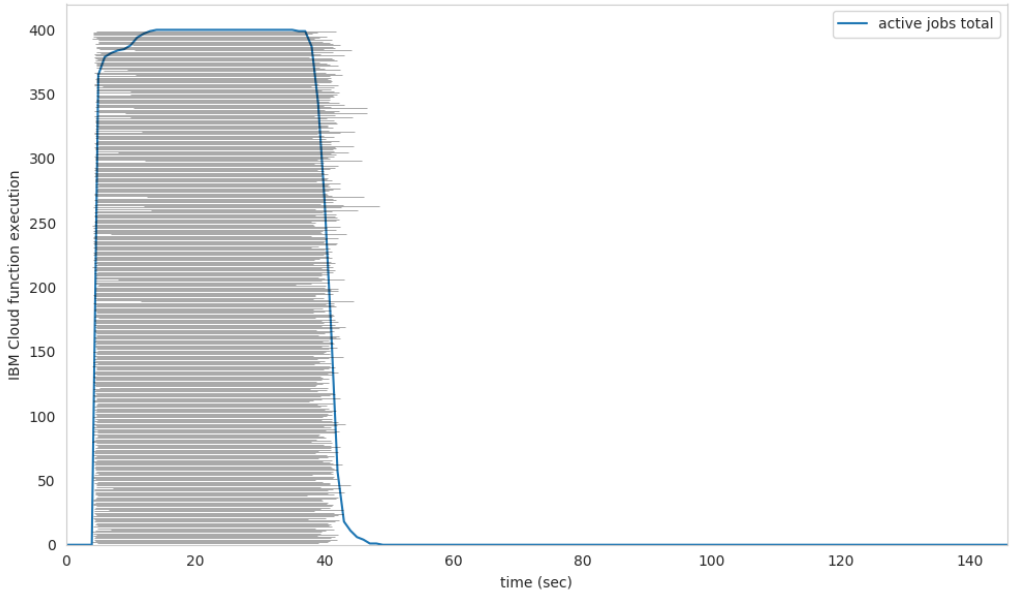


Figura 7. Exemple 2 de l'activació de funcions amb Lithops-cloud

Com es pot observar la majoria de càrrega de treball es produeix en els segons posteriors a l'inici de l'execució, el qual permet que la càrrega de treball treballi al màxim rendiment des d'un inici. També es pot observar com algunes de les funcions s'activen posteriorment, fet que retarda durant uns segons la finalització de l'execució, però no de forma considerable.

2.6 Algoritmes genètics al núvol

Per aprofitar els beneficis del Cloud executant algoritmes genètics hi ha diferents aproximacions que es poden utilitzar [12].

2.6.1 Algoritmes genètics Paral·lels

La versió d'algoritmes genètics paral·lels es basa en la idea d'utilitzar **vàries execucions** d'algoritmes genètics per a donar solució a un problema. Totes aquestes execucions tracten de resoldre el **mateix** problema i un cop finalitzen la seva execució **retornen el millor individu** que han produït, de tots aquests individus retornats es selecciona el millor de tots i és la solució al problema.

Aquest model s'anomena normalment **"model d'illes"**, ja que aquestes execucions s'executen sense dependre unes de les altres com si estiguessin aïllades.

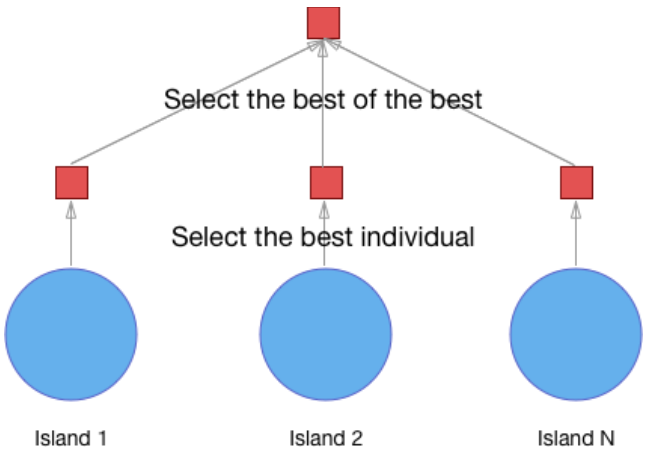


Figura 8. Algoritmes genètics paral·lels

2.6.2 Algoritmes genètics distribuïts

La versió d'algoritmes genètics de forma paral·lela realment és com la versió d'algoritmes genètics paral·lels però executant-se en diferents màquines al mateix temps en comptes d'una sola. Aprofitant que s'executen en **diferents màquines** s'executa el mateix model d'illes en cada màquina per a poder obtenir el millor individu de cada màquina i un cop s'ha trobat el millor individu es posen en comparació aquests millors i es selecciona la millor solució. Aquesta versió també s'anomena **"model d'arxipèlags"**, ja que junta diversos models d'illes formant un arxipèlag.

Principalment està pensat per aprofitar diferents màquines per poder fer encara més iteracions de l'algoritme en paral·lel, però també es pot donar el cas que una sola màquina no sigui capaç d'emmagatzemar la quantitat d'individus necessaris per a trobar la solució i el que es fa és tenir diversos espais de cerca, un en cada màquina per a que cadascuna es centri en una part de la solució.

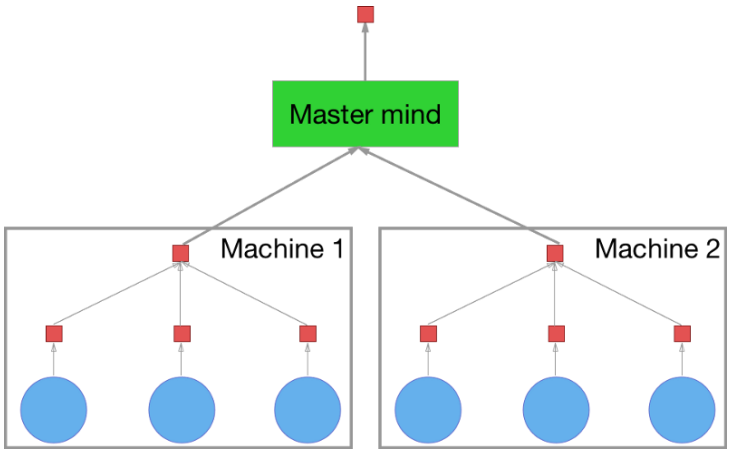


Figura 9. Algoritmes genètics distribuïts

2.6.3 Altres possibles aproximacions

A més a més de les versions proposades existeixen altres tipus de paral·lelitzacions i variacions que es poden realitzar per aprofitar encara més l'execució al núvol.

Per aprofitar encara més l'execució del model d'illes es poden **variar els paràmetres d'entrada** de cada illa (probabilitat de mutació, nombre de seleccionats, mida de la població...) o **l'algoritme de selecció i/o creuament** per a generar també individus més variats entre cada illa i d'aquesta forma tenir una diversitat més generalitzada i poder trobar solucions òptimes de forma més senzilla.

Una altra aproximació és la **d'intercanviar** alguns dels **millors individus** de cada illa quan es realitza la selecció d'individus o la generació de la nova població. D'aquesta manera podem aportar diversitat a la població de les diferents illes i trobar de forma més senzilla la solució general en comptes d'un màxim o mínim local. S'ha de tenir en compte de no fer aquest procés molt aviat, ja que serviria de molt poc en haver-hi molta diversitat al principi de l'execució; ni molt tard, ja que les solucions ja estan prou consolidades i la variació amb altres individus podria ajudar a allunyar-se de la solució òptima.

Per últim també es pot **paral·lelitzar** cada execució de l'algoritme, en concret **la funció de fitness** o **la funció de creuament d'individus**. Es poden paral·lelitzar aquestes funcions per a **millorar el temps d'execució de l'algoritme**, ja que són funcions que s'han d'executar molts cops durant l'execució de l'algoritme, i si són complexes de realitzar depenent del problema fer-les de forma paral·lela pot incrementar l'eficiència d'una forma considerable.

3 Disseny

En aquesta secció s'explica com està distribuïda l'arquitectura del sistema Serverless segons l'aproximació de Lithops i quin és el flux d'execució que segueix l'aplicació i els seus components [13].

3.1 Arquitectura Serverless amb Lithops

L'arquitectura Serverless està composta per diferents components que seran uns o altres depenent de cada aplicació. En aquest cas els components principals que s'utilitzen són el client, el Cloud Computing i l'Object Storage, on cada un d'aquests components s'encarrega d'una tasca diferent per a l'execució al Cloud.

- El **Client o Client Driver** és l'encarregat de l'execució de l'algoritme principal i de l'execució de les crides remotes al Cloud. En aquest cas és un ordinador personal des del qual s'inicia primerament tota la preparació per fer les crides remotes, prepara els Workers per a la seva execució i defineix un dipòsit d'emmagatzematge dintre del Object Storage per a guardar dades necessàries per a l'execució de cada Worker, els resultats de les tasques o altres dades per a l'execució.
- El **Cloud Computing o Computing Backend** és l'encarregat de fer l'execució de les tasques que s'han especificat, en aquest cas consisteix en el Cloud Functions del proveïdor IBM. Descàrrega les diferents dades necessàries per a la seva execució guardades al Object Storage, realitza les tasques que té encomanades i finalment guarda els diferents resultats al Object Storage per a processar-los al acabar.
- El **Object Storage o Storage Backend** és l'encarregat d'emmagatzemar tota la informació temporal per a l'execució al Cloud en un dipòsit. Tant el client driver com el Cloud computing interactuen amb ell per a emmagatzemar o descàrregar tota la informació necessària durant l'execució, i és un mètode de comunicació entre el client i els Workers per a saber per exemple la finalització de l'execució. La manera d'emmagatzemar les dades al Object Storage és utilitzant la serialització[13], ja que s'ha de mantenir l'estructura i l'estat de les dades en tot moment per a la seva posterior utilització.

3.2 Flux d'execució de Lithops

El flux d'execució de la llibreria Lithops es realitza a través dels components mencionats durant l'arquitectura, és a dir, a través del Client, del Cloud Computing i de l'Object Storage. El seu flux de treball per a realitzar qualsevol execució és el següent:

1. **Inicialització.** El Client, a l'inicialitzar l'execució, configura els diferents treballadors, defineix el dipòsit d'emmagatzematge al Cloud per a emmagatzemar les dades/paràmetres necessàries per a l'execució i fa inicialitzacions necessàries dels diferents treballs a realitzar.
2. **Invocació.** Un cop finalitzada la inicialització, el client comença a realitzar les diferents crides remotes per a la seva execució al Cloud Computing a través de la funció map(). Si alguna d'aquestes crides és denegada o fallida, es reintenta un altre cop després d'un període aleatori d'espera.

[13] La serialització és el procés de convertir un objecte en una cadena de bytes per a guardar un objecte o transmetre'l a memòria, una base de dades o un fitxer. El seu propòsit és el de conservar l'estat de l'objecte per a la seva recuperació posterior.

3. **Execució.** Un cop realitzades les invocacions el Cloud Computing realitza la descàrrega de les dades/paràmetres del Object Storage que ha preparat el Client, i comença a executar les tasques que té encomanades. Al finalitzar guarda el resultat de forma serialitzada al Object Storage per a la seva utilització durant la reducció.
4. **Reducció.** Un cop ha finalitzat l'execució de les tasques és crida a la funció reduce() des del Cloud Computing o des del Client (segons el mode desitjat), que crea una altra crida remota o diverses crides remotes pel processament dels resultats. Aquesta crida descàrrega del Object Storage els resultats de les anteriors crides, fa els càlculs que hagi de realitzar al Cloud Computing i guarda el resultat al Object Storage.
5. **Finalització.** Un cop la reducció ha acabat, el client descàrrega del Object Storage el resultat computat i finalitza la seva execució o neteja les dades residuals del dipòsit del Object Storage.

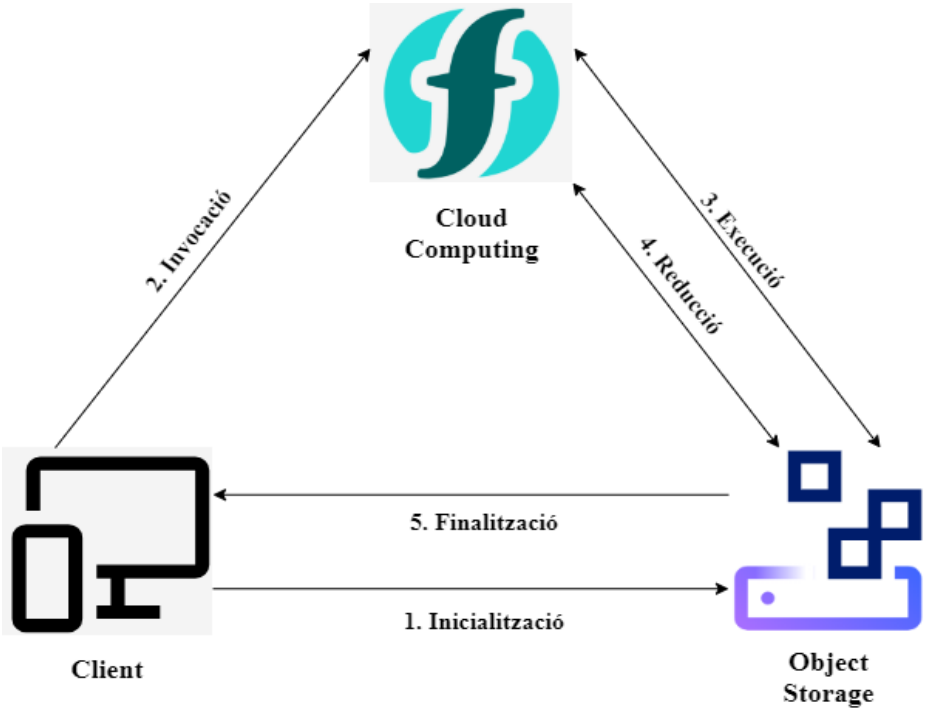


Figura 10. Flux execució arquitectura Serverless amb Lithops

L'espera a la finalització de cada tasca té dues formes de realitzar-se. La primera de totes utilitza la publicació de missatges per avisar que una tasca ha sigut realitzada. La segona forma és utilitzant l'enquesta periòdica al Object Storage per comprovar si el resultat de la tasca ja ha sigut computat.

A més a més, la utilització de Lithops comporta una sèrie de costos afegits. El temps "perdut" quan una funció finalitza i es rep que ha finalitzat i el seu resultat és un exemple, que degut a la congestió de la xarxa i la serialització i deserialització d'informació pot incrementar més. Un altre exemple pot ser la redundància de dades en les funcions, quan es repeteix la mateixa dada múltiples cops per a diferents funcions, Lithops no disposa de cap forma de reduir la redundància i repeteix la mateixa dada per totes les funcions en comptes d'assignar una sola dada per a totes aquelles funcions on es repeteix, per tant perdem espai de memòria i reduïm l'eficiència de l'intercanvi d'informació entre el client i l'Object Storage. Per últim, al realitzar múltiples peticions al proveïdor des d'una sola màquina es pot denegar la petició, una mesura de protecció d'atacs de denegació de servei, Lithops reintenta la petició amb un petit retard que ralentitza la inicialització de l'execució.

4 Implementació

En aquest punt es parla sobre la implementació adoptada per resoldre el problema del TSP amb algoritmes genètics i estudiar el seu funcionament amb la tecnologia Serverless.

4.1 Especificacions del projecte

Per la realització del projecte s'han seguit una sèrie d'especificacions bàsiques per a l'elaboració i utilització d'aquest.

La tasca principal de l'algoritme genètic és la de retornar la millor solució que ha trobat durant la seva execució, aquesta solució està composta pel camí que proposa l'individu i el seu valor fitness.

La tasca principal de la part Serverless és la d'executar, segons les especificacions que es proporcionen, una certa quantitat de vegades l'algoritme genètic de forma paral·lela utilitzant el sistema Serverless, recopilar totes les solucions que es proposen i retornar la millor solució que s'ha pogut trobar, donant-se aquesta com a solució del problema.

Al tractar-se d'un projecte de recopilació de resultats per la seva anàlisi posterior no s'ha tingut en compte cap mena d'interacció amb l'usuari. Tot i així, per facilitar algun possible canvi de valors de variables s'ha creat un fitxer de constants per facilitar-los. En aquest fitxer es poden controlar paràmetres de l'execució Serverless com el nom del dipòsit i la paral·lelització, o paràmetres específics de l'algorisme genètic i la mida del problema TSP.

A més a més, totes les configuracions de credencials o proveïdor Cloud es realitzen en un fitxer addicional de configuració de Lithops per a facilitar-ne el canvi entre projectes o usuaris.

4.2 Estructura del projecte

El projecte està desenvolupat en Python per aprofitar el framework Lithops per facilitar la utilització de la tecnologia Serverless, utilitza el sistema de control de versions GIT per mantenir un control dels canvis realitzats al projecte i les seves diferents versions i segueix l'estructura següent:

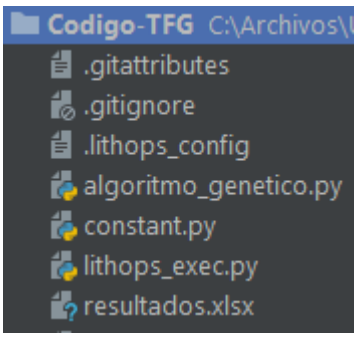


Figura 11. Estructura del projecte

- El fitxer **.gitattributes** és un fitxer de text que dona atributs als noms de les rutes del repositori.
- El fitxer **.gitignore** permet afegir fitxers que han de ser ignorats a l'hora de controlar diferents versions del projecte, com pot ser la carpeta cache de Python o fitxers de proves.
- El fitxer **.lithops_config** és un fitxer de configuració del mòdul Lithops per a configurar el proveïdor Cloud, les diferents credencials de cada servei que s'utilitza o alguns paràmetres de configuració de les funcions. Es pot especificar a nivell de

projecte, com en aquest cas, a través d'un objecte a l'hora de realitzar la inicialització de Lithops o de forma global a la carpeta local de l'usuari.
- El fitxer **algoritmo_genetico.py** conté tota la implementació de la resolució del problema TSP a través d'algoritmes genètics, el qual s'explica en els següents apartats.
- El fitxer **constant.py** defineix algunes variables globals per a facilitar la modificació de variables que s'utilitzen en molts apartats del projecte, com poden ser el nombre de ciutats/mida del problema TSP, la ciutat d'origen del problema TSP, el nombre d'individus de cada generació, el nombre de generacions que s'ha d'executar l'algoritme genètic, el nombre de seleccionats durant la selecció, el nombre de participants en cada Tour Selection, la probabilitat de mutació d'un gen, el nom del dipòsit d'emmagatzematge al núvol, el nombre de Workers a utilitzar i la quantitat d'execucions que s'han de realitzar de l'algoritme.
- El fitxer **lithops_exec.py** conté l'algoritme principal d'execució al Cloud juntament amb la implementació de la preparació de dades per a l'execució i la definició de la funció map() i de la funció reduce().
- Per últim, el fitxer **resultados.xlsx** emmagatzema els resultats obtinguts durant l'estudi de l'execució d'algoritmes genètics al núvol, amb els temps d'execució per a diferents nombres de Workers i nombre d'execucions.

Per a la realització del projecte a més a més s'han utilitzat algunes llibreries Python externes:

- **Numpy:** per a la creació i tractament de matrius i arrays de forma eficient en Python.
- **Random:** per a la generació de nombres aleatoris.
- **Time:** per a l'obtenció del temps del sistema per calcular els temps d'execució.
- **Lithops:** per a l'execució de l'algoritme genètic al núvol.
- **Pickle:** per a la serialització d'objectes pel núvol.
- **Openpyxl:** per a escriure els resultats en fitxers excel per a la seva anàlisi posterior.

4.3 Solució implementada

Per a parlar sobre la solució que s'ha implementat primerament es parlarà de com s'ha solucionat el problema del TSP amb algoritmes genètics i després de tota la part d'execució de l'algoritme amb el Cloud. Depenent de si és necessari s'adjunta una aproximació en forma de pseudocodi per facilitar la comprensió de l'explicació de la solució proposada.

4.3.1 Implementació del TSP amb algoritmes genètics

El TSP té algunes restriccions per a poder ser resolt i que s'han de tenir en compte durant la implementació. Aquestes restriccions en aquest cas són que cada ciutat només es pot visitar un cop, cada solució proposada ha de fer un recorregut per totes les ciutats i s'ha de començar i acabar per la ciutat d'origen. Qualsevol solució proposada que compleixi aquestes condicions és considerada com a vàlida.

En aquest cas, el problema del TSP amb algoritmes genètics consisteix en un problema de minimització de fitness, és a dir, el que es pretén és reduir el màxim possible el fitness de les solucions, perquè les millors solucions seran aquelles amb el recorregut més curt.

4.3.1.1 Generador de matrius TSP

Per a poder simplificar el procés de generació de mapes TSP s'ha creat un generador de matrius aleatori. D'aquesta forma es poden crear mapes TSP de qualsevol mida sense cap inconvenient.

Per a fer aquest procés s'ha decidit que els problemes TSP generats serien simètrics, però es podrien fer asimètrics de forma senzilla amb una petita modificació. A més a més, s'ha afegit la possibilitat de la creació de connexions entre ciutats impossibles (pes del camí -1), per afegir una mica de restricció a l'hora de generar solucions; i la possibilitat d'especificar la distància màxima que pot haver-hi entre cada ciutat.

Al tractar-se d'una matriu simètrica, la qual no es pot viatjar sobre la mateixa ciutat, la diagonal de la matriu està formada per zeros, i només s'ha de generar la meitat de la matriu, ja que al ser simètrica si s'accedeix a la posició [3, 2], és el mateix que accedir a la [2, 3], i per tant el número generat per a la primera posició serà el mateix que pel de la segona.

Cada posició del mapa correspon al pes del camí d'anar d'una ciutat a una altra.

```
[ 0  4 10 -1]
[ 4  0  4  7]
[10  4  0  2]
[-1  7  2  0]
```

Figura 12. Exemple de mapa TSP.

Per exemple, en el mapa de la Figura 10, on hi ha 4 ciutats, anar des de la ciutat 0 (Fila 0) a la 2 (columna 2) té un cost de 10 unitats, anar des de la ciutat 2 (Fila 2) a la 0 (Columna 0) té també un cost de 10, ja que és el mateix al tractar-se d'un problema simètric. Anar des de la ciutat 3 a la 0 té un camí impossible de recórrer.

4.3.1.2 Algoritme genètic

L'estructura d'un algoritme genètic segueix sempre els mateixos passos, i és la següent:

```
algoritme_genetic(mapa_tsp, n_cromosomes, n_gens, n_generacions,
n_seleccionats, prob_mutacio, n_tour, ciutat_origen):
  Comprovació_parametres()
  poblacio_nova = inicialitzacióPoblacio(n_cromosomes, n_gens,
ciutat_origen)
  for i in range(n_generacions):
    // Calcular el fitness de la població, ordenar-la i trobar el millor
    poblacio_nova = calcular_fitness(poblacio_nova, mapa_tsp)
    poblacio_actual = població_nova.sort()
    millor_solució = poblacio_actual[0]
    // Fer la selecció, el creuament i la mutació de la nova població
    poblacio_nova = seleccio(poblacio_actual, n_seleccionats, n_tour)
    poblacio_nova = creuament(poblacio_nova, n_cromosomes)
    població_nova = mutació(poblacio_nova, prob_mutacio)
  return(millor_solucio)
```

Pseudocodi 1. Funcionament general d'un algoritme genètic

Com es pot observar els passos a seguir són inicialitzar la població el primer cop per a tenir individus i durant un nombre de generacions calcular el fitness de la població, seleccionar els individus, fer el creuament dels seleccionats i mutar els nous individus. Un cop finalitzat tot el procés es retorna el millor individu de l'última generació.

4.3.1.3 Inicialització de la població

Per a guardar la informació de la població s'ha creat una estructura **cromosoma** la qual té la **informació genètica de l'individu**, la qual es tracta d'una array on el primer gen representa la ciutat d'origen i la resta representen les diferents ciutats i l'ordre en què es visiten; i el **valor del fitness de l'individu,** per no haver de calcular-lo cada cop que es necessiti. D'aquesta forma s'agilitza l'accés de la informació de l'individu tant del fitness com de la solució que proposa.

Per inicialitzar la població simplement el que es fa és crear un individu amb un nombre de gens igual al nombre de ciutats que s'especifica de forma ordenada, i s'intercanvia la ciutat d'origen amb la primera posició. Fins que no es té el nombre d'individus desitjats el que es fa és crear nous individus a partir de l'individu anterior fent una barreja de tots els gens excepte el primer, i d'aquesta forma s'obté els individus aleatoris. A més, cada gen és únic representant una sola ciutat i satisfent la restricció de només visitar un cop cada ciutat.

4.3.1.4 Càlcul del Fitness

Cada vegada que es calcula el fitness el que es fa és crear l'estructura cromosoma mencionada en la inicialització, on el cromosoma és la informació genètica que es don i el fitness es calcula amb la funció de fitness. Per a calcular el fitness es necessita la informació genètica de cada individu, a més del mapa TSP del problema. És en aquest punt on es tracta la restricció que s'ha de tornar a la ciutat d'origen. El **fitness mínim** que ha de tenir un individu és **1**, al tractar-se d'un problema de minimització. Com hi ha camins entre ciutats els quals no es poden utilitzar (pes del camí -1) el fitness en aquests individus serà de -1, fet que significarà que és un individu amb una solució no vàlida.

El fitness es calculà seguint l'ordre de les ciutats que proposa cada individu. El que es fa és accedir al gen actual i al següent, on el gen actual representa la fila a accedir i la posició següent la columna a accedir del mapa. Un cop s'arriba a l'última posició, el gen actual representa l'última posició i el següent gen la posició d'origen.

```
fitness(individu, mapa):
  fitness = 0
  for i in range(len(individu) - 1):
    valor = mapa[individu[i], individu[i + 1]]
    if valor != -1: fitness = fitness + valor
    else: //Hem trobat un camí impossible, fitness impossible
      fitness = -1
      break
  if fitness > 0: //Calcular tornar del final al origen
    valor = mapa[individu[-1], individu[0]]
    if valor != -1: fitness = fitness + valor
    else: fitness = -1
  return fitness
```

Pseudocodi 2. Funció de calcular fitness d'un individu

```
calcular_fitness(població, mapa):
  població_nova = []
  for i in range(len(poblacio)):
    individu = població[i].copy()
    poblacio_nova.append(Cromosoma(individu, fitness(individu, mapa)))
  return població_nova
```

Pseudocodi 3. Funció de calcular fitness de tots els individus

4.3.1.5 Algoritme de selecció

Per a seleccionar individus existeixen diversos mètodes, però el mètode escollit ha estat el **Tournament Selection** [14].

El Tournament Selection o Torneig de Selecció es basa en el principi **d'escollir el millor individu d'una quantitat N d'individus aleatoris** de la població, així successivament fins que s'arriba a la quantitat de seleccionats desitjada. Fa servir el principi d'elitisme al seleccionar el millor individu, però també fa servir el principi d'aleatorietat al fer competir individus aleatoris. Si el nombre d'individus del Tour és molt petit, hi ha moltes probabilitats que solucions dolentes siguin seleccionades, si el nombre del Tour és molt gran, hi ha moltes probabilitats d'elegir sempre al mateix individu al tenir millor fitness.

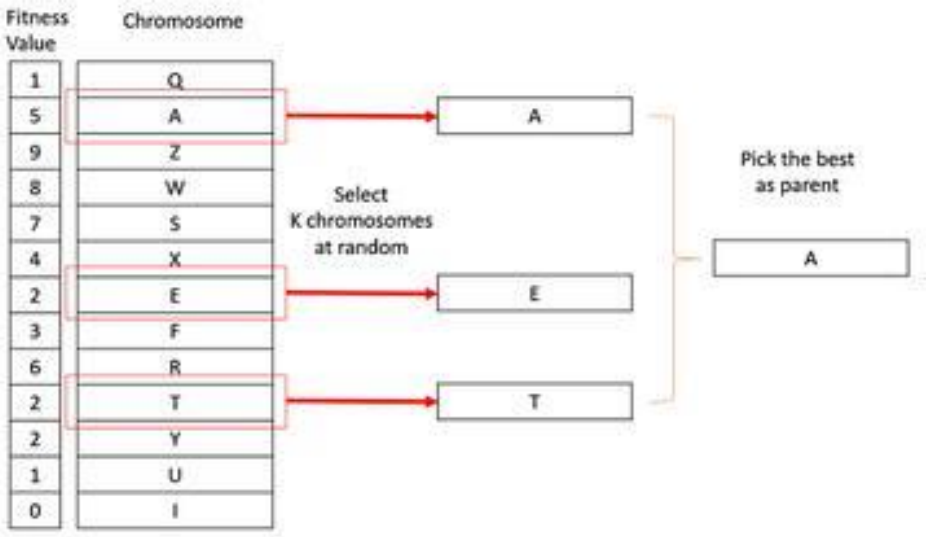


Figura 13. Representació del Tournament Selection

En aquest cas la implementació que s'ha realitzat ha sigut la següent:

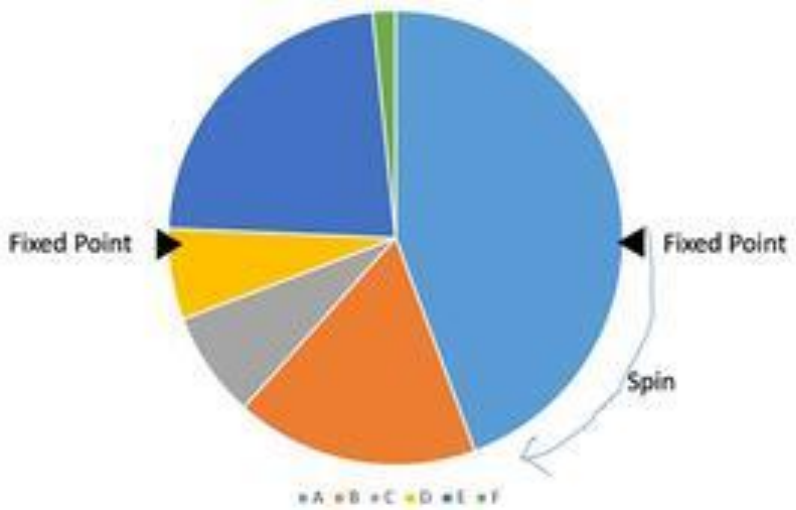
```
Tournament_selection(població, n_seleccionats, n_tour):
  població_selec = []
  for i in range(n_seleccionats):
    seleccionat = població[random(len(poblacio))]
    for j in range(1, n_tour):
      competidor = població[random(len(poblacio))]
      //Fem competir el millor individu fins al moment amb un altre
      //individu aleatori. També intercanviem si el fitness no és vàlid
```

```
        if(seleccionat.fitness < 1) || (0 < competidor.fitness <
seleccionat.fitness):
            seleccionat = competidor
      població_selec.append(seleccionat)
   return població_selec
```

Pseudocodi 4. Funció de Tournament Selection

A més a més, també s'ha implementat el mètode de selecció Stochastic Universal Sample [15] o SUS, però no s'ha acabat utilitzant degut a la gran ineficiència amb el TSP. El problema principal és que les distancies entre fitness en el TSP acaben sent molt grans, llavors s'acaba seleccionant individus on la solució que proposen no és ideal per a evolucionar molts cops; fent que es necessitin moltes més generacions i individus per a poder tenir solucions òptimes. Per això, molts cops s'allunya de solucions òptimes o és molt ineficient comparat amb el Tournament Selection.

La idea principal del SUS és que es té una "roda" partida en trossos. Cada tros representa la proporció segons el fitness de cada individu, llavors els individus amb millor fitness tenen més proporció que la resta. Segons la quantitat de seleccionats que es vulgui el que es fa es fer tantes marques com seleccionats es vulgui, on les marques tenen una distància equitativa. Un cop es té les marques i les proporcions s'agafa un nombre aleatori entre el 0 i la primera marca, el qual és el punt d'inici. A partir d'aquest punt d'inici i les marques el que es fa és seleccionar individus segons si han sigut "marcats", és a dir, es suma a les marques el punt d'inici i es sabrà si un individu ha sigut marcat o no. Com el TSP és un problema de minimització s'han de normalitzar els valors del fitness per a què els que menor fitness tenen siguin els que tinguin major proporció.



Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

Figura 14. Representació Universal Stochastic Sampling

4.3.1.6 Algoritme de creuament

Per realitzar la nova població el que es fa és fer un creuament de dos pares per formar dos fills en cada iteració. D'aquesta forma cada fill tindrà informació genètica dels pares com si s'haguessin creuat. Per realitzar aquest procés s'ha de tenir en compte que el TSP té les restriccions que la ciutat d'origen s'ha de mantenir la mateixa, totes les ciutats han de ser recorregudes i no pot visitar-se la mateixa ciutat dos cops. Per a complir totes aquestes restriccions s'ha d'utilitzar un algoritme de creuament que mantingui aquestes condicions, per això l'algorisme que s'ha implementat ha sigut el del **Partially Mapped Crossover o PMX** [15]**.**

El PMX es basa en la idea d'utilitzar dos punts de partició aleatoris per partir cada pare en 3. La part central és la que es conserva totalment de cada pare cap als seus fills, el primer fill rep la part central del primer pare i el segon fill la part central del segon pare. A partir d'aquí el primer fill rep la primera i última part del segon pare, tots els gens que encara no s'han utilitzat a la mateixa posició i el mateix pel segon fill però amb els gens del primer pare. Finalment, es col·loquen en els llocs restants els gens que falten de forma ordenada segons l'ordre en què apareguin en cada pare amb el seu fill corresponent.

$$P_1 = (3\ 4\ 8 \mid 2\ 7\ 1 \mid 6\ 5)$$
$$P_2 = (4\ 2\ 5 \mid 1\ 6\ 8 \mid 3\ 7)$$

Figura 15. Pares amb els punts de partició amb PMX

$$O_1 = (\times\ \times\ \times \mid 1\ 6\ 8 \mid \times\ \times)$$
$$O_2 = (\times\ \times\ \times \mid 2\ 7\ 1 \mid \times\ \times)$$

Figura 16. Representació del primera pas del PMX

$$O_1 = (3\ 4\ \times \mid 1\ 6\ 8 \mid \times\ 5),$$
$$O_2 = (4\ \times\ 5 \mid 2\ 7\ 1 \mid 3\ \times)$$

Figura 17. Representació del segon pas del PMX

$$O_1 = (3\ 4\ 2 \mid 1\ 6\ 8 \mid 7\ 5)$$

Figura 18. Primer fill generat amb PMX

$$O_2 = (4\ 8\ 5 \mid 2\ 7\ 1 \mid 3\ 6)$$

Figura 19. Segon fill generat amb PMX

4.3.1.7 Mutació de la població

Per completar la implementació de l'algoritme genètic hi ha la mutació dels individus per "simular" les mutacions que es poden produir en l'evolució normal i **oferir diversitat** d'individus.

Per a oferir la mutació als diferents individus amb el problema del TSP, i mantenir les restriccions de començar per una ciutat d'origen en específic i visitar totes les ciutats únicament un cop, no es pot utilitzar el mètode de la negació de gens o generar una ciutat aleatòria per un gen, s'ha de fer l'intercanvi de gens per a simular la mutació.

Per a fer aquest intercanvi s'ha implementat el codi següent:

```
mutacio(població, prob_mutacio):
  poblacio_mutada = []
  for i in range(len(poblacio)):
    individu = població[i]
    for j in range(1, len(individu)):
      rand = random()
      if rand < prob_mutacio: //Comprovar si hi ha mutació
        pos = random(1, len(individu))
        tmp = individu[j]
        individu[j] = individu[pos]
        individu[pos] = tmp
    població_mutada.append(individu)
  return poblacio_mutada
```

Pseudocodi 5. Funció de mutació d'individus

S'ha de tenir en compte també que si la probabilitat de mutació és molt baixa la diversitat d'individus que ofereix la mutació potser no serà suficient per trobar altres solucions que no siguin màxims o mínims locals. Si la probabilitat és molt alta pot ser que les solucions s'allunyin de solucions ideals al mutar tan fàcilment els seus gens i empitjorar totes les solucions òptimes. Una probabilitat de mutació ideal normalment es situa entre el 5% i el 15%, però pot variar segons cada problema.

4.3.2 Implementació de l'execució Serverless

Un cop es té tota la implementació de l'algoritme genètic, la implementació de l'execució al Cloud amb tecnologia Serverless és molt senzilla de realitzar amb el framework **Lithops**. En aquest cas s'ha utilitzat l'API de computació de **Futures**, juntament l'API **Storage** per a emmagatzemar dades.

Primerament es realitza la configuració del proveïdor Cloud on s'executa el codi, i després tota la implementació del codi. En aquest cas el que es fa primerament és executar una funció asíncrona per preparar possibles dades per l'execució; després es crida als diferents Workers[14] per fer les execucions al Cloud; i per últim es recopilen tots els resultats de les funcions per a treballar-los i retornar el resultat final.

[14] Un Worker és el procés que s'utilitza per fer l'execució al cloud. Es pot encarregar de fer una o vàries execucions, ja que es reparteixen la càrrega de treball per agilitzar l'execució.

4.3.2.1 Funció call_async()

Aquesta funció és la que s'utilitza per **preparar** les diferents **dades necessàries** per a l'execució al núvol. En aquest cas les dades necessàries per a aquest problema és el **mapa TSP**, per a poder reduir la mida de les dades que es passa a cada funció, ja que el mapa TSP en mides grans pot arribar a tenir una mida considerable. Simplement el que fa és pujar el mapa a l'emmagatzematge del Cloud.

4.3.2.2 Funció map()

La funció map() és la que executa cada Worker a l'hora de fer l'execució al núvol. En aquest cas el que fa és **descarregar el mapa TSP** del problema i **executar l'algoritme genètic** amb els paràmetres que s'especifiquen per aquella execució. Un cop finalitza la seva execució retorna el resultat a la funció reduce() per unir tots els resultats obtinguts. Es crida a la funció map tantes vegades com execucions al servidor es vol realitzar, és a dir, si es vol executar l'algoritme 1000 cops, es cridarà a la funció 1000 cops.

4.3.2.3 Funció reduce()

La funció reduce() és l'encarregada de **reunir tots els resultats** i **tractar-los** per a retornar un resultat final. En aquest cas el seu propòsit és el de comprovar totes les solucions que cada execució ha retornat, comparar-les i retornar la millor o millors solucions que s'han obtingut. Si es proposen diferents solucions amb el mateix fitness, es retornen totes per mostrar les possibles solucions que hi pot haver al problema.

Un aspecte a tenir en compte és que la funció reduce només pot retornar estructures pròpies de Python (Llistes, diccionaris, primitius...), llavors no es pot retornar l'estructura pròpia cromosoma, ja que no és serialitzable. La solució a aquest problema ha estat transformar els resultats que retornen les funcions a un diccionari de Python, que retornarà el cromosoma i el seu fitness.

4.3.2.4 Implementació conjunta

Finalment totes les funcions treballant de forma conjunta proporcionen l'execució de l'algoritme genètic amb la tecnologia Serverless. L'aproximació bàsica queda de la següent manera:

```
main():
   fexec = lithops.ServerlessExecutor(workers=const.n_workers)
   tsp = ag.matrixTSP(const.N_CIUTATS, const.RANG)
   fexec.call_async(carregar_mapa, tsp)
   //Preparar els paràmetres que rebrà cada execució, en aquest cas totes
   //les execucions rebran els mateixos paràmetres però es poden variar
   iterdata = []
   for i in range(cons.N_EXECUCIONS):
      iterdata.append(parametres_ag)
   fexec.map_reduce(map_function=exec_ag, map_iterdata=iterdata,
reduce_function=proposar_solucio, include_modules=["constant",
"algoritme_genetic"])
   solució = fexec.get_result()
   fexec.clean() //Neteja de dades residuals
```

Pseudocodi 6. Funcionament general execució amb Lithops

4.4 Avaluació de la solució proposada

Per comprovar el correcte funcionament de la solució implementada s'han realitzat una sèrie de comprovacions. Les comprovacions són les següents:

- Comprovació de paràmetres correctes i incorrectes tant en l'execució general, com en les diferents funcions de forma individual.
- Generació correcta de mapes TSP amb diferents mides, rangs, representació de la simetria de forma correcta, 0s en la diagonal i generació de camins impossibles (pes -1).
- Inicialització de diferents individus de forma diversa, sempre concordant el nombre d'individus a generar, la mida d'aquests individus i la ciutat d'origen amb els paràmetres especificats. A més a més, comprovar que els individus tinguessin adreces de memòria diferents per la seva correcta modificació posterior.
- Càlcul del fitness de cada individu de forma correcta, segons el camí de cada individu o si té un camí impossible retornar fitness -1. També s'ha comprovat que faci el càlcul del fitness de tots els individus i que es guardi de forma correcta en una llista d'estructura Cromosoma.
- Selecció d'individus a l'atzar de forma correcta i diversa amb l'algoritme de selecció Tournament Selection per a la realització de "tornejos" i que el millor individu d'aquests a l'atzar sigui seleccionat. A més que el nombre de seleccionats siguin els especificats.
- Comprovar que el creuament entre els individus es faci de forma correcta amb el procés de creuament PMX, mantenint la informació dels pares entre els dos punts aleatoris, i la correcta posició de la resta de gens segons la resta de gens utilitzats.
- Comprovar que la mutació de gens es realitzi de forma correcta, mirant que els gens s'intercanviïn amb una posició aleatòria amb el gen a mutar i que mai s'intercanviï amb la ciutat d'origen.
- Comprovar que l'algoritme genètic s'executi amb els paràmetres especificats, produint solucions d'acord amb el problema i retorni la solució mínima general o solucions òptimes que s'apropin en la majoria de casos.
- Comprovar que les dades necessàries per a l'execució al Cloud es carreguin i descarreguin de forma correcta.
- Comprovar que el nombre d'execucions realitzades correspongui amb el nombre de solucions retornades, a més que d'aquestes solucions retornades es proposi com a solució la millor de totes.
- Per últim comprovar que l'eliminació d'arxius residuals al Cloud es faci de forma correcta al final de cada execució.

5 Resultats obtinguts

En aquest apartat es comparen els diferents resultats obtinguts durant l'execució de l'algoritme en l'àmbit local i utilitzant la tecnologia Serverless. A més, s'analitzen els diferents resultats obtinguts per a diferents nombres d'execucions, quantitat de processadors que es reserven per a l'execució i les solucions obtingudes segons el nombre d'execucions realitzades.

5.1 Metodologia utilitzada

Els resultats que s'han obtingut en aquest cas són, primerament, els **temps d'execució** de l'algoritme per a diferents nombres de processadors (Workers) i diferents quantitats d'execucions a realitzar; i, posteriorment, el nombre de **solucions que s'apropen** a la solució general d'un problema en específic. D'aquesta forma podem observar la **millora del rendiment** en forma de Speedup[15] segons el nombre de processadors i execucions; i la qualitat de les solucions que se'ns proposen amb diferent nombre d'execucions per a veure realment si és necessari tantes execucions.

Per a poder facilitar la tasca d'obtenció de resultats s'utilitza el mòdul de Python Openpyxl per a la manipulació de fitxers Excel amb Python. D'aquesta forma tots els resultats obtinguts de les diferents execucions es guarden de forma estructurada en un fitxer Excel de resultats per després facilitar la tasca d'anàlisis i tractament de dades.

Per automatitzar la tasca d'executar diferents cops l'algoritme amb **diferent nombre d'execucions i nombre de processadors** (Workers) a utilitzar, s'ha creat un petit procés automàtic a través de dues llistes, on a la primera s'especifica el nombre de processadors a utilitzar, i a la segona el nombre d'execucions a realitzar. D'aquesta forma per cada nombre de processadors es realitzen els diferents nombres d'execucions i es guarden al fitxer, i així successivament fins a finalitzar amb totes les execucions amb tots els processadors.

En aquest cas s'ha estudiat el rendiment tant de forma local (si ho executem a una màquina pròpia) com amb la utilització de servidors del proveïdor IBM.

De forma local s'ha estudiat com en una màquina de 8 cores[16] 16 threads[17] funcionant a una freqüència de 3,6 GHz (Ryzen 7 3700X) s'executa l'algoritme. El nombre de Workers a estudiar ha sigut 1, 8 i 16 Workers; i les execucions a realitzar han sigut 1, 5, 10, 25, 50, 100, 200, 300, 400.

Utilitzant els servidors d'IBM el nombre de Workers a estudiar ha sigut 1, 8, 16, 25, 50, 100, 200, 300, 400, 500, 600; i amb un nombre d'execucions mínim de 1, 5, 10, 25, 50, 100, 200, 300, 400 i en alguns casos també amb 500, 600, 700, 800, 900, 1000, 1100, 1200.

Per a garantir que els resultats es mantenen de forma estable s'ha executat cada nombre d'execucions 5 cops i s'ha fet la mitja de les 5 execucions. D'aquesta forma si hi ha hagut algun increment o decrement del rendiment puntual es suavitza al resultat final.

[15] El Speedup és la comparació de rendiment entre la versió seqüencial de l'execució i la versió paral·lela per a comprovar quantes vegades millora. En aquest cas la versió seqüencial correspon a 1 Worker, i la versió paral·lela a més d'un Worker.

[16] Un core correspon a un processador físic en tota la seva funcionalitat, és a dir, té tots els recursos hardware necessaris d'un processador per a executar les instruccions. Ens permet l'execució de diversos programes de forma independent en cada core.

[17] Un thread correspon a la compartició d'un processador entre diversos programes, és a dir, quan el processador està executant diversos programes de forma concurrent quan un programa està llegint un altre programa pot aprofitar per escriure a memòria. D'aquesta forma comparteixen l'execució del processador.

5.1.1 Problemes durant l'obtenció de resultats

Durant l'obtenció de resultats han sorgit alguns problemes que han dificultat la tasca de recopilació que són interessants de comentar.

El primer de tots i que a la llarga ha acabat afectant més ha sigut els temps d'execució per a alguns nombres de Workers i nombres d'execucions. En alguns casos el temps que tardava a fer una sola execució arribava a ser de fins a 20 minuts en el cas del servidor i de 26 minuts en el cas de la màquina local, provocant que l'obtenció de resultats fos molt lenta.

El segon i que juntament amb el primer també ha dificultat molt la tasca és que a vegades la inicialització de les execucions al servidor era més lenta de l'habitual, provocant que finalment saltes un timeout del temps d'execució màxim i pares tota l'obtenció de resultats. Això podia ser degut a l'inici en fred dels servidors o a la congestió d'aquests durant les hores puntes depenent de la zona horària, ja que no hi ha la mateixa quantitat d'usuaris de 9 am a 5 pm que de 12 pm a 9 am per exemple. L'error de Timeout que es mostrava era el següent:

```
MAPA SUBIDO
-------------------------------------
INICIO NÚMERO WORKERS: 16
DATOS PREPARADOS - 16 WORKERS 400 EJECUCIONES
ExecutorID afc5bf-1 | JobID M000 - There was an exception - Activation ID: 8907406fb378407887406fb378007805
Exception: TimeoutError - The function did not run as expected.

Process finished with exit code 1
```

Figura 20. Timeout execució per inicialització lenta.

Similar al problema anterior a vegades quan porta moltes execucions en un espai de temps curt sense parar, salta un error HTTP 423 (Too Many Requests) informant què s'han fet moltes peticions al recurs del mapa TSP. Passa perquè tots els proveïdors tenen un nombre màxim de peticions depenent del pla de facturació que es té, i en aquest cas, al tractar-se d'un pla gratuït, és molt més fàcil arribar-hi. L'error és el següent:

```
ExecutorID fad2bd-1 | JobID M030 - There was an exception - Activation ID: 67dd56926fc847b09d56926fc897b0f5
Traceback (most recent call last):
  File "/action/lithops/worker/jobrunner.py", line 296, in run
  File "C:/Archivos/Uni/TFG/Codigo-TFG/lithops_exec.py", line 32, in exec_ag
    tsp = pickle.loads(storage.get_object(bucket=const.BUCKET_NAME, key='mapa.txt'))
  File "/action/lithops/storage/storage.py", line 88, in get_object
  File "/action/lithops/storage/backends/ibm_cos/ibm_cos.py", line 147, in get_object
  File "/usr/local/lib/python3.8/site-packages/ibm_botocore/client.py", line 357, in _api_call
  File "/usr/local/lib/python3.8/site-packages/ibm_botocore/client.py", line 662, in _make_api_call
  File "/usr/local/lib/python3.8/site-packages/ibm_botocore/client.py", line 682, in _make_request
  File "/usr/local/lib/python3.8/site-packages/ibm_botocore/endpoint.py", line 102, in make_request
  File "/usr/local/lib/python3.8/site-packages/ibm_botocore/endpoint.py", line 132, in _send_request
  File "/usr/local/lib/python3.8/site-packages/ibm_botocore/endpoint.py", line 115, in create_request
  File "/usr/local/lib/python3.8/site-packages/ibm_botocore/hooks.py", line 356, in emit
  File "/usr/local/lib/python3.8/site-packages/ibm_botocore/hooks.py", line 228, in emit
  File "/usr/local/lib/python3.8/site-packages/ibm_botocore/hooks.py", line 211, in _emit
  File "/usr/local/lib/python3.8/site-packages/ibm_botocore/signers.py", line 90, in handler
  File "/usr/local/lib/python3.8/site-packages/ibm_botocore/signers.py", line 154, in sign
  File "/usr/local/lib/python3.8/site-packages/ibm_botocore/signers.py", line 234, in get_auth_instance
  File "/usr/local/lib/python3.8/site-packages/ibm_botocore/credentials.py", line 2912, in get_frozen_credentials
  File "/usr/local/lib/python3.8/site-packages/ibm_botocore/credentials.py", line 2454, in get_token
  File "/usr/local/lib/python3.8/site-packages/ibm_botocore/credentials.py", line 2650, in _get_initial_token
  File "/usr/local/lib/python3.8/site-packages/ibm_botocore/credentials.py", line 2539, in _default_auth_function
ibm_botocore.exceptions.CredentialRetrievalError: Error when retrieving credentials from https://iam.cloud.ibm.com/identity/token: HttpCode(429) - Retrieval of tokens from server failed.
```

Figura 21. Error Too Many Requests del mapa TSP.

Per últim, al realitzar moltes peticions de forma simultània durant la invocació de Workers a vegades ha saltat la protecció d'atacs DOS[18], que en aquest cas el que feia simplement era tombar momentàniament algunes d'aquestes peticions de connexió remota,

[18] Un atac DOS és un atac de denegació de servei, que el que fa és no permetré accedir a recursos a base de fer moltes peticions al mateix temps i saturar el servidor que allotja el recurs.

i, posteriorment després d'un retard aleatori, es tornava a reintentar. Aquest rebutjament podia provocar retards en l'execució total de l'algoritme.

5.2 Resultats obtinguts

Tots els resultats s'han extret amb una mida del mapa TSP de 50 ciutats, i amb uns paràmetres de l'algoritme genètic de 100 individus per generació, 100 generacions a executar, amb 1/3 dels individus seleccionats, un 20% dels individus durant el Tour Selection i una probabilitat de mutació del 10%. D'aquesta forma durant totes les execucions es conserven els paràmetres inicials i es pot veure la seva evolució al modificar el nombre d'execucions a realitzar o el nombre de Workers a utilitzar.

5.2.1 Resultats amb mode local

En el mode local s'han obtingut resultats per a 1, 8 i 16 Workers. I el nombre d'execucions han sigut de 1, 5, 10, 25, 50, 100, 200, 300 i 400 execucions.

Els temps d'execució en segons obtinguts en mode local són els següents:

Nombre Execucions	1	5	10	25	50	100	200	300	400
1 Worker (s)	5,63	19,50	37,81	91,79	182,20	364,29	806,33	1237,26	1651,12
8 Workers (s)	5,55	5,70	9,71	17,82	30,64	56,12	107,81	161,05	212,72
16 Workers (s)	5,97	6,11	7,46	12,59	22,35	39,45	73,86	108,56	142,63

Taula 2. Resultats Temps Execució (s) en mode local

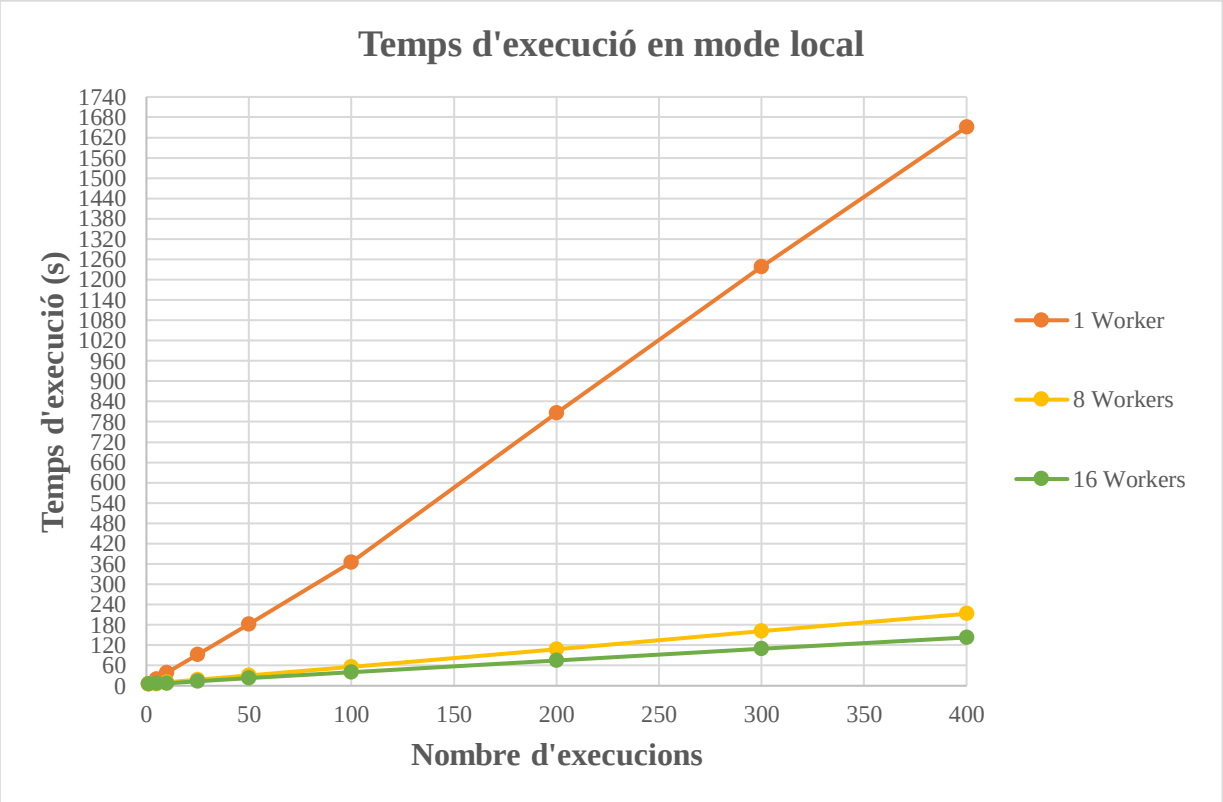


Figura 22. Gràfica temps d'execució en mode local

Com s'observa, els temps d'execució de treballar de forma seqüencial a treballar de forma paral·lela es redueix de forma considerable, passant de tardar 1651 segons amb la versió seqüencial a 142 segons de forma paral·lela i 16 Workers. També s'ha de comentar, encara que no estigui relacionat, que tot i utilitzar 16 Threads lògics de processament la millora que genera respecte a 8 Cores físics no és molt elevada, veient-se així que els Cores tenen molta més millora que els Threads al disposar de tot el hardware necessari per a l'execució paral·lela.

A més a més, amb els resultats anteriors es pot obtenir el Speedup de passar d'utilitzar 1 Worker a més d'un. Aquest Speedup consisteix en el càlcul de dividir el pitjor temps entre el temps millorat, és a dir, Temps d'1 Worker / Temps X Workers. D'aquesta forma es pot observar la millora de rendiment entre diferents Workers. Els valors obtinguts serien els següents:

Nombre Execucions	1	5	10	25	50	100	200	300	400
Speedup 8 Workers	1,01	3,42	3,89	5,15	5,95	6,49	7,48	7,68	7,76
Speedup 16 Workers	0,94	3,19	5,07	7,29	8,15	9,23	10,92	11,40	11,58

Taula 3. Speedup execució local

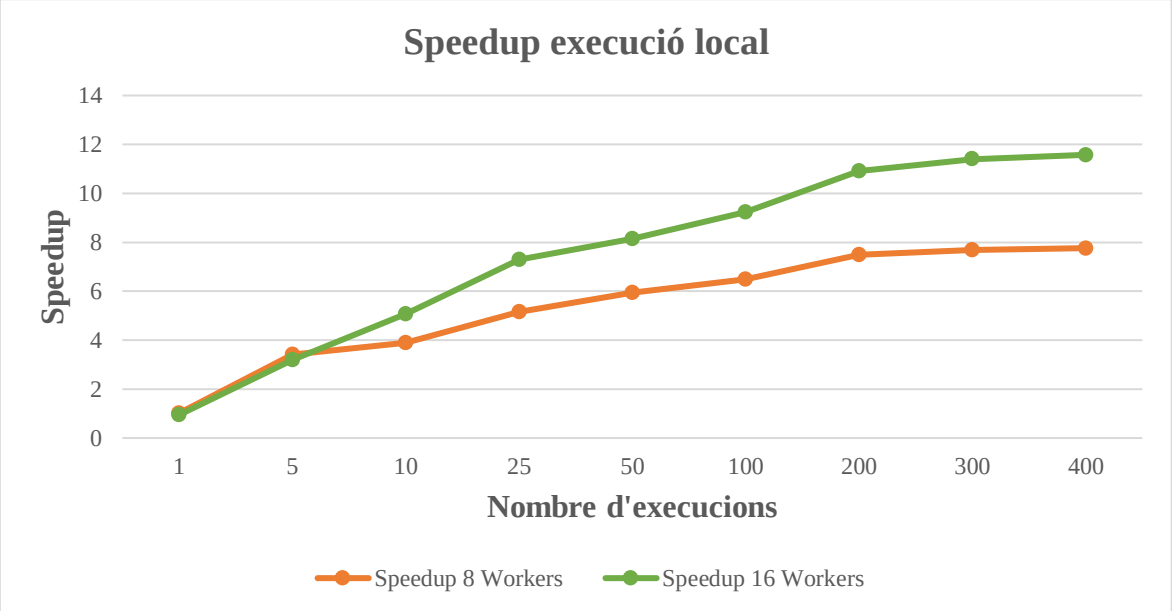


Figura 23. Gràfica Speedup execució local

Com es pot observar la millora de Speedup entre 8 Workers i 16 Workers s'eleva segons s'augmenta el nombre d'execucions a realitzar, fins a començar-se a estabilitzar aquesta millora a l'arribar a 200 execucions. A més, la millora de Speedup de passar de 8 a 16 Workers no es duplica com s'esperaria al tenir el doble de Workers, degut al comentat anteriorment, que no tenen el mateix rendiment 8 Cores o processadors físics que 16 Threads o processadors lògics.

5.2.2 Resultats amb servidor

En el mode d'execució al servidor hi ha resultats per a 1, 8, 16, 25, 50, 100, 200, 300, 400, 500 i 600 Workers. S'ha estudiat per a 1, 5, 10, 25, 50, 100, 200, 300, 400 execucions per a tots els Workers i, a més a més, per alguns Workers s'ha treballat amb 500, 600, 700, 800, 900, 1000, 1100 i 1200 execucions. S'han decidit utilitzar aquest nombre d'execucions i de Workers per a veure com evoluciona l'escalabilitat de l'aplicació de forma progressiva i amb quantitats elevades; i també per comparar posteriorment el mode local i el Serverless.

Els temps d'execució, en segons, obtinguts al servidor han sigut els següents:

Nombre Execucions	1	5	10	25	50	100	200	300	400
1 Worker (s)	8,02	15,14	28,35	67,53	89,13	158,30	252,55	799,00	1343,18
8 Workers (s)	7,59	15,48	28,49	71,24	93,18	143,15	185,66	783,41	1042,01
16 Workers (s)	8,41	15,58	29,07	67,71	87,59	96,53	180,28	264,30	1158,92
25 Workers (s)	8,10	16,75	31,10	75,61	95,70	100,98	190,83	400,53	557,57
50 Workers (s)	7,30	16,01	29,30	67,85	90,90	92,56	157,30	382,00	422,15
100 Workers (s)	7,93	15,83	29,54	71,89	88,85	90,65	135,64	162,94	272,88
200 Workers (s)	12,45	16,48	29,16	69,97	88,13	91,45	134,57	147,85	176,83
300 Workers (s)	8,50	16,24	29,25	68,11	90,27	92,97	139,87	127,54	137,02
400 Workers (s)	7,98	16,62	30,31	67,01	87,93	90,34	136,76	133,09	134,04
500 Workers (s)	7,92	17,80	31,62	78,28	95,94	96,55	108,03	106,77	117,55
600 Workers (s)	8,91	11,43	19,35	47,71	57,91	89,87	150,60	129,25	143,94

Nombre Execucions	500	600	700	800	900	1000	1100	1200
1 Worker (s)	-	-	-	-	-	-	-	-
8 Workers (s)	-	-	-	-	-	-	-	-
16 Workers (s)	-	-	-	-	-	-	-	-
25 Workers (s)	667,63	812,05	-	-	-	-	-	-
50 Workers (s)	529,53	626,93	621,96	868,26	-	-	-	-
100 Workers (s)	280,21	332,78	382,85	417,77	480,99	543,22	593,21	685,34
200 Workers (s)	189,39	255,97	278,72	294,43	321,47	356,84	411,14	521,16
300 Workers (s)	157,25	171,59	197,90	226,64	258,86	285,42	291,53	346,77
400 Workers (s)	142,87	152,77	166,43	170,72	200,53	213,94	244,12	319,38
500 Workers (s)	135,02	142,81	146,47	160,51	172,56	180,79	206,19	231,06
600 Workers (s)	142,22	136,95	185,52	185,53	189,24	190,02	189,24	201,72

Taula 4. Temps d'execució (s) en mode Serverless

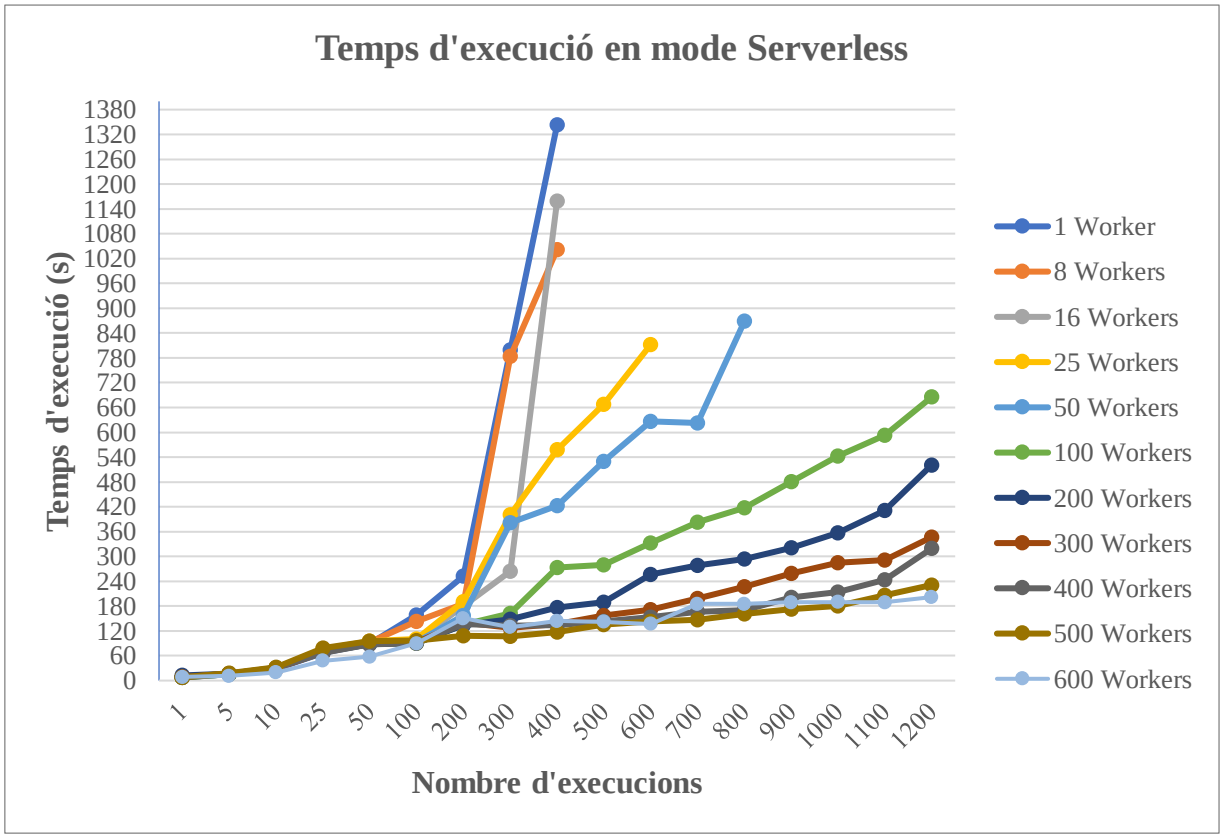


Figura 24. Gràfica temps d'execució en mode Serverless

Com es pot observar la diferència d'anar augmentant el nombre de Workers amb els quals es treballa és molt notable a mesura que es va augmentant els Workers. Tant notable és que amb un únic Worker el màxim d'execucions a les quals s'ha arribat és de 400 execucions. Això és degut a que cada execució té un time out màxim del mateix servidor per mantenir la filosofia Serverless d'execucions curtes i espontànies.

A mesura que s'escala el nombre de Workers, augmenten el nombre d'execucions màximes que es poden realitzar fins que s'arriba a 100 Workers i 1200 execucions, que és el límit d'execucions que s'ha decidit estudiar degut a que ja es veu clarament la tendència de les dades.

La tendència dels resultats quan hi ha poques execucions resulta interessant, ja que els resultats es mantenen igual tant si s'utilitza 1 Worker com 600 fins que no s'arriba a les 100 execucions. Això pot ser degut a que els sistemes Serverless tenen un inici en fred, és a dir, quan s'inicia l'execució hi ha un període on no s'està treballant al 100% de la capacitat, ja que molts cops els servidors que s'utilitzen s'han d'iniciar des de zero per realitzar la tasca. Això fa que fins que no s'arribi a la capacitat màxima de treball els resultats d'utilitzar pocs Workers o molts no es vegin realment beneficiats a l'augmentar-ne el nombre.

També es pot observar que la tendència dels Workers i la seva millora en el temps d'execució va lligat al nombre d'execucions que s'hagin de realitzar. Si el nombre d'execucions és baix, com per exemple 400, el nombre de Workers necessaris no té per què ser 600, amb 200 Workers es poden obtenir resultats similars sense utilitzar tants recursos. La tendència segueix passant quan s'augmenta el nombre d'execucions, per exemple al realitzar 1200 execucions hi ha aproximadament 120 segons de diferència entre utilitzar 300/400 Workers o 500/600, per tant es poden ajustar el nombre de recursos a 300 o 500 Workers en comptes de 400 o 600. Si s'estudiés aquesta tendència per a més Workers es

veuria com al final la utilització de tants Workers és innecessària, ja que s'obtindrien pràcticament els mateixos resultats amb menys recursos. És a dir, ajustar el nombre de Workers a uns paràmetres correctes pot comportar un estalvi significatiu en els recursos necessaris per a l'execució.

Com en el cas del mode local, també s'ha estudiat la millora de rendiment en forma de Speedup. Com no ha sigut possible obtenir resultats de tots els Workers per a totes les execucions s'ha dividit aquest estudi en 2. El primer estudia el Speedup de tots els Workers respecte als resultats obtinguts amb 1 sol Worker. El segon estudia l'Speedup de tots els Workers que han pogut arribar a 1200 execucions.

Els resultats obtinguts de Speedup comparant tots els Workers han sigut els següents:

Nombre Execucions	1	5	10	25	50	100	200	300	400
8 Workers	1,06	0,98	1,00	0,95	0,96	1,11	1,36	1,02	1,29
16 Workers	0,95	0,97	0,98	1,00	1,02	1,64	1,40	3,02	1,16
25 Workers	0,99	0,90	0,91	0,89	0,93	1,57	1,32	1,99	2,41
50 Workers	1,10	0,95	0,97	1,00	0,98	1,71	1,61	2,09	3,18
100 Workers	1,01	0,96	0,96	0,94	1,00	1,75	1,86	4,90	4,92
200 Workers	0,64	0,92	0,97	0,97	1,01	1,73	1,88	5,40	7,60
300 Workers	0,94	0,93	0,97	0,99	0,99	1,70	1,81	6,26	9,80
400 Workers	1,00	0,91	0,94	1,01	1,01	1,75	1,85	6,00	10,02
500 Workers	1,01	0,85	0,90	0,86	0,93	1,64	2,34	7,48	11,43
600 Workers	0,90	1,32	1,47	1,42	1,54	1,76	1,68	6,18	9,33

Taula 5. Speedup execució Serverless amb 1 Worker

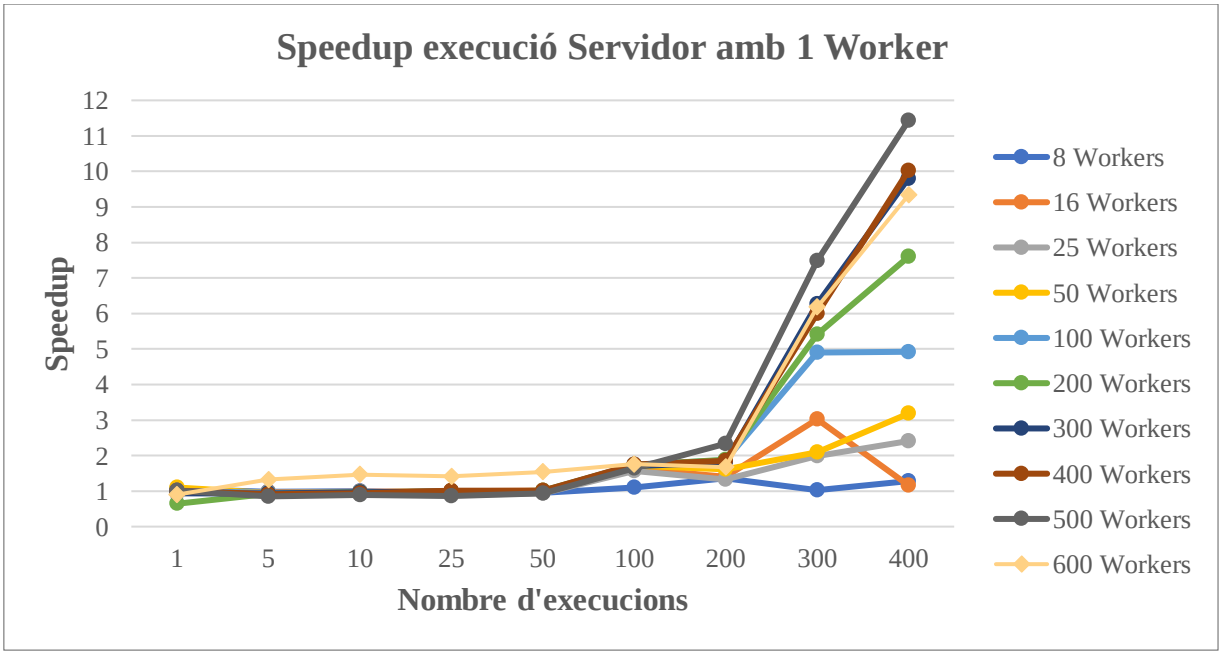


Figura 25. Gràfica Speedup execució Serverless amb 1 Worker

Com en la gràfica del temps d'execució es pot observar com la millora de rendiment d'utilitzar 1 Worker o 600 no augmenta fins que no es comencen a realitzar 100 execucions i fent-se molt més notable a les 300 execucions. També es torna a observar com a les 400 execucions és indiferent la utilització de 300 Workers o 600, ja que pràcticament obtenen resultats idèntics. Tot i així, quan es realitzen 400 execucions el rendiment entre utilitzar 1 Worker o més arriba a augmentar fins a 12 vegades, veient-se clarament com sí que millora de forma considerable els temps d'execució.

Els resultats obtinguts de Speedup dels Workers que han arribat a 1200 execucions són els següents, sent 100 Workers el pitjor temps obtingut:

Nombre Execucions	1	5	10	25	50	100	200	300	400
200 Workers	0,64	0,96	1,01	1,03	1,01	0,99	1,01	1,10	1,54
300 Workers	0,93	0,97	1,01	1,06	0,98	0,98	0,97	1,28	1,99
400 Workers	0,99	0,95	0,97	1,07	1,01	1,00	0,99	1,22	2,04
500 Workers	1,00	0,89	0,93	0,92	0,93	0,94	1,26	1,53	2,32
600 Workers	0,89	1,38	1,53	1,51	1,53	1,01	0,90	1,26	1,90

Nombre Execucions	500	600	700	800	900	1000	1100	1200
200 Workers	1,48	1,30	1,37	1,42	1,50	1,52	1,44	1,32
300 Workers	1,78	1,94	1,93	1,84	1,86	1,90	2,03	1,98
400 Workers	1,96	2,18	2,30	2,45	2,40	2,54	2,43	2,15
500 Workers	2,08	2,33	2,61	2,60	2,79	3,00	2,88	2,97
600 Workers	1,97	2,43	2,06	2,25	2,54	2,86	3,13	3,40

Taula 6. Speedup execució Serverless amb 100 Workers

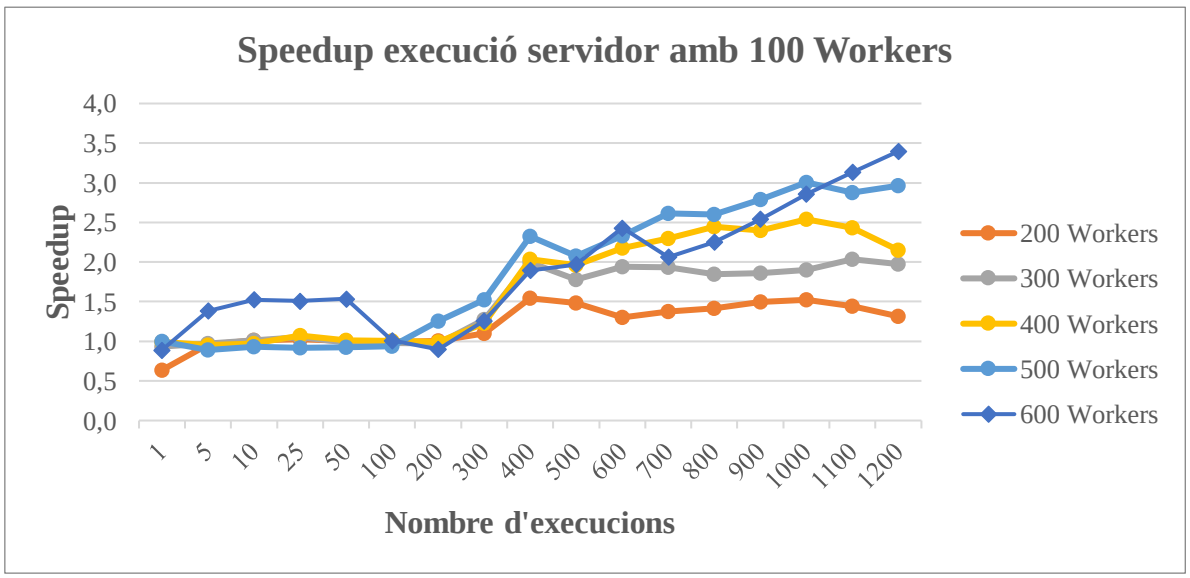


Figura 26. Gràfica Speedup execució Serverless amb 100 Workers

Com es pot observar, al principi de l'execució es veu més clarament l'inici en fred dels servidors, produint retards i variacions en els temps d'execució de poques execucions. Però quan comença a agafar el rendiment òptim es veu un increment notable entre les 200 i 400 execucions, fins a estabilitzar-se de forma progressiva segons el nombre de Workers i, fins i tot, començar a decrementar-se quantes més execucions han de realitzar. Tot i així, s'observa una millora de rendiment de fins a 3,5 vegades utilitzant 600 Workers respecte als 100 Workers de referencia, veient-se així que sí s'utilitzen més Workers la millora de rendiment si que és bastant significativa.

5.2.3 Comparació entre local i servidor

Per a comprovar realment si l'execució Serverless aporta alguna millora respecte a l'execució local s'han comparat els 8 i 16 Workers locals amb els 100, 200 i 600 Workers Serverless. D'aquesta forma es comparen el nombre de Workers amb els temps d'execució semblants i el nombre de Workers màxims en cada mode.

Els resultats de temps d'execució obtinguts han sigut els següents:

Nombre Execucions	1	5	10	25	50	100	200	300	400
8 Workers - Local	5,55	5,7	9,71	17,82	30,64	56,12	107,81	161,05	212,72
16 Workers - Local	5,97	6,11	7,46	12,59	22,35	39,45	73,86	108,56	142,63
100 Workers - Server	7,93	15,83	29,54	71,89	88,85	90,65	135,64	162,94	272,88
200 Workers - Server	12,45	16,48	29,16	69,97	88,13	91,45	134,57	147,85	176,83
600 Workers - Server	8,91	11,43	19,35	47,71	57,91	89,87	150,60	129,25	143,94

Nombre Execucions	500	600	700	800	900	1000	1100	1200
8 Workers - Local	253,71	307,23	356,24	411,90	458,52	522,24	552,41	617,50
16 Workers - Local	170,75	205,51	239,14	269,76	305,65	337,14	371,42	406,96
100 Workers - Server	280,21	332,78	382,85	417,77	480,99	543,22	593,21	685,34
200 Workers - Server	189,39	255,97	278,72	294,43	321,47	356,84	411,14	521,16
600 Workers - Server	142,22	136,95	185,52	185,53	189,24	190,02	189,24	201,72

Taula 7. Temps execució Local vs Serverless

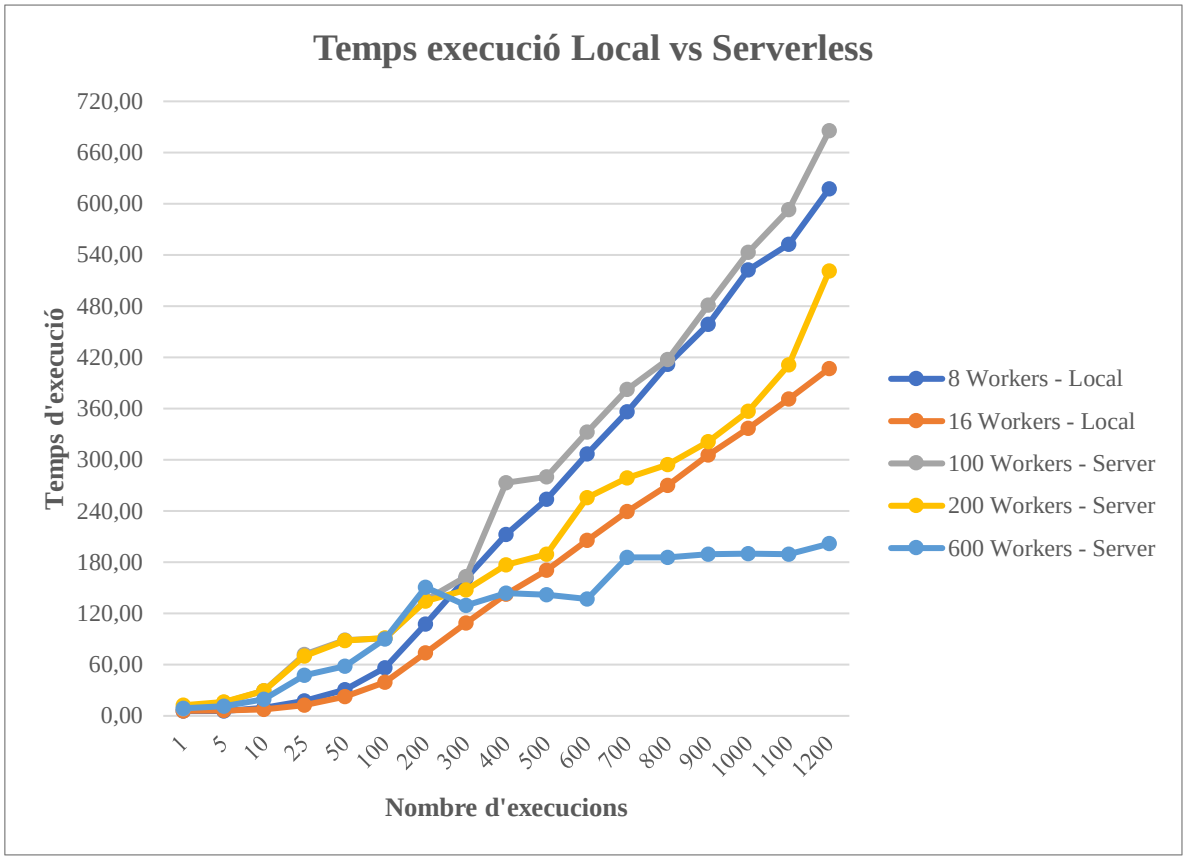


Figura 27. Gràfica temps execució Local vs Serverless

Com es pot observar en els resultats els temps d'execució que més s'assimilen als 8 Workers locals són els 100 Workers del mode Serverless, i dels 16 Workers locals són els 200 Workers Serverless.

El mode local es veu molt beneficiat del retard en l'execució inicial del mode Serverless, ja que com es pot observar fins que no s'arriben a les 300-400 execucions el mode local és més eficient que el mode Serverless. A partir de les 300-400 execucions el mode Serverless s'equiparà en molts casos al mode local, però a l'observar els 600 Workers Serverless es pot observar com en aquest cas hi comença a haver una gran diferència de rendiment respecte els 16 Workers locals.

Un altre punt a destacar és la constància de creixement lineal del mode local, el qual no ofereix inestabilitats en els seus resultats a mesura que es va augmentant el nombre d'execucions. En canvi el mode Serverless ens ofereix resultats no lineals veient-se així les variacions de rendiment, velocitat de xarxa o temps d'engegada dels servidors i de la computació.

També s'obtenen els Speedups de comparar 8 vs 100 Workers i 16 vs 200 Workers:

Nombre Execucions	1	5	10	25	50	100	200	300	400
8 vs 100 Workers	0,70	0,36	0,33	0,25	0,34	0,62	0,79	0,99	0,78
16 vs 200 Workers	0,48	0,37	0,26	0,18	0,25	0,43	0,55	0,73	0,81
8 vs 600 Workers	0,62	0,50	0,50	0,37	0,53	0,62	0,72	1,25	1,48
16 vs 600 Workers	0,67	0,53	0,39	0,26	0,39	0,44	0,49	0,84	0,99

Nombre Execucions	500	600	700	800	900	1000	1100	1200
8 vs 100 Workers	0,91	0,92	0,93	0,99	0,95	0,96	0,93	0,90
16 vs 200 Workers	0,90	0,80	0,86	0,92	0,95	0,94	0,90	0,78
8 vs 600 Workers	1,78	2,24	1,92	2,22	2,42	2,75	2,92	3,06
16 vs 600 Workers	1,20	1,50	1,29	1,45	1,62	1,77	1,96	2,02

Taula 8. Speedup Local vs Serverless

Figura 28. Gràfica Speedup 8 vs 100 Workers

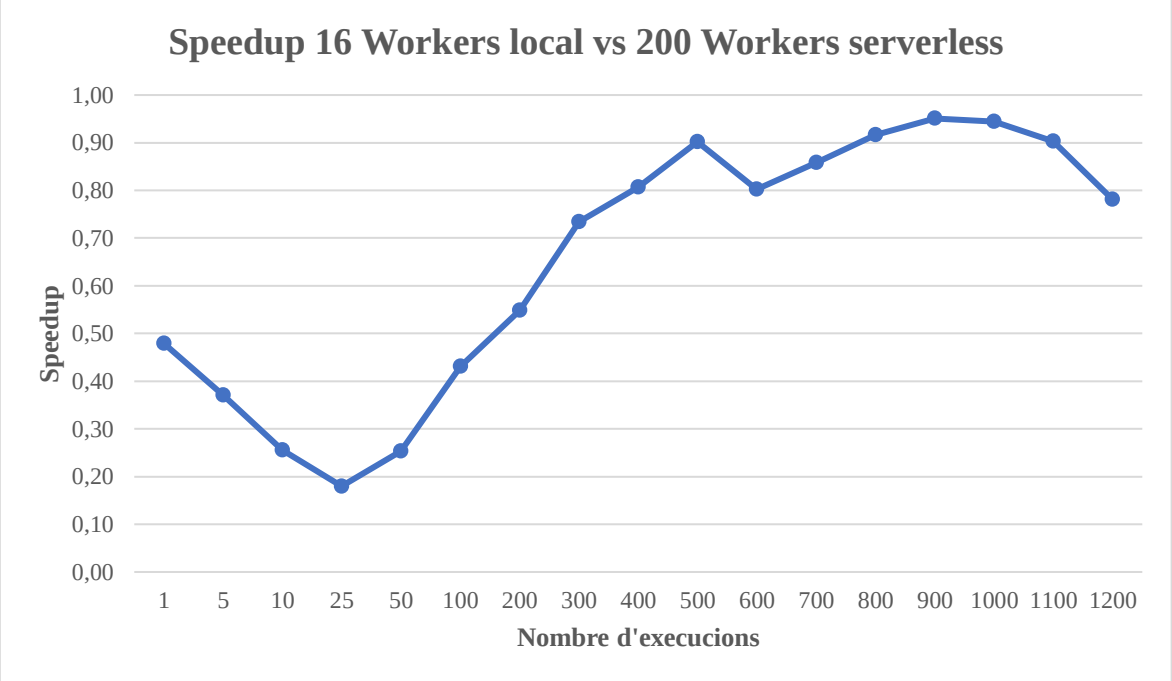


Figura 29. Gràfica Speedup 16 vs 200 Workers

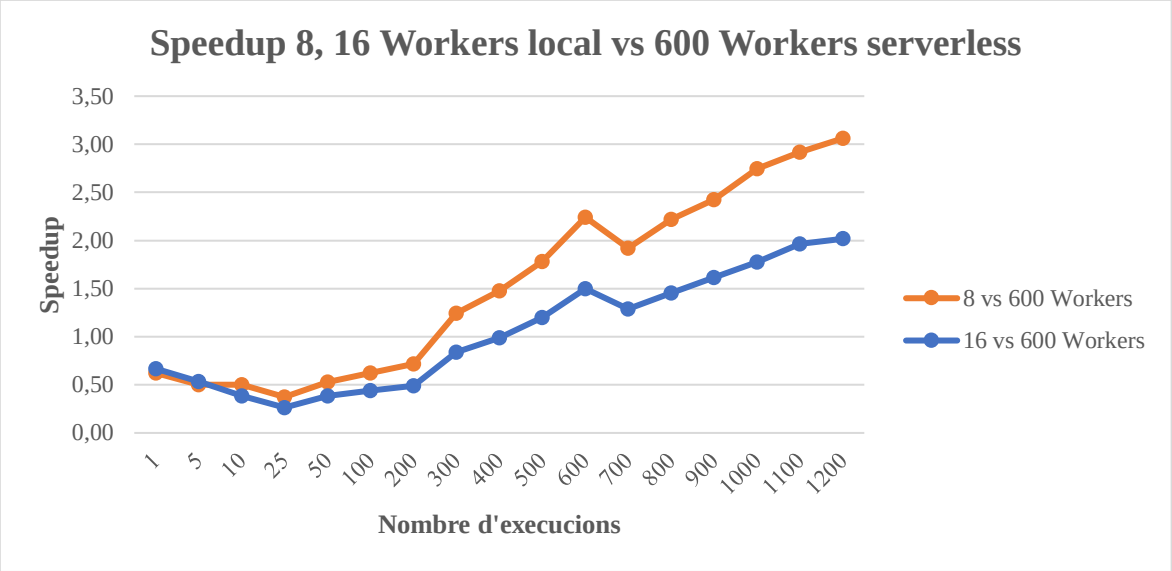


Figura 30. Gràfica Speedup 8-16 vs 600 Workers

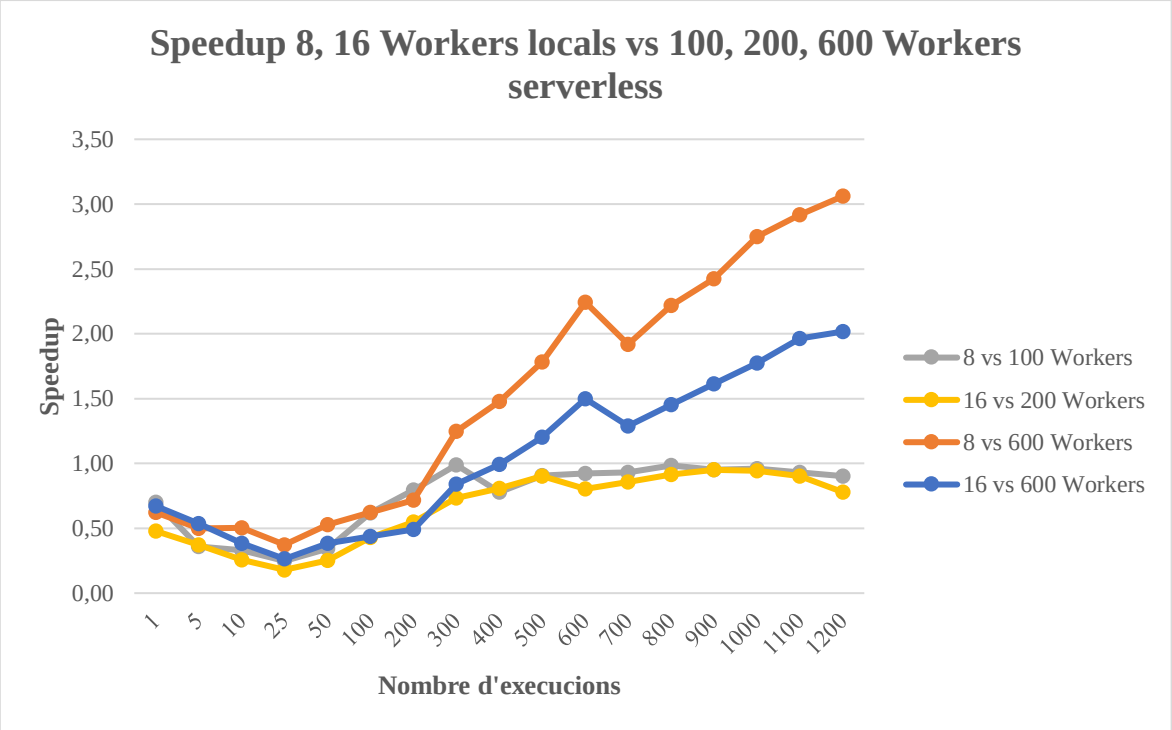


Figura 31. Gràfica Speedup local vs Serverless

Com es pot observar en els gràfics de Speedup anteriors 8 Workers locals mostren resultats similars a 100 Workers Serverless a l'apropar-se finalment a un Speedup d'1. El mateix passa amb 16 Workers locals i 200 Workers Serverless. També s'aprecia l'inici en fred de la tecnologia Serverless, que ofereix un rendiment molt per sota que el mode local fins que no s'arriba a les 300-400 execucions.

Amb 600 Workers s'observa el mateix comportament d'inici en fred, que a l'arribar a les 300-400 execucions sobrepassa els resultats del mode local obtenint fins a 2 i 3 vegades més rendiment que els 16 Workers locals.

5.2.4 Millora de les solucions proposades

Alguns problemes resolts amb algoritmes genètics tenen tendència a trobar-se amb mínims o màxims locals i, per a trobar la solució òptima general, es necessita executar l'algoritme diverses vegades per aproximar encara més la solució. En aquest cas, el TSP és un problema que té aquesta tendència, i per això s'ha comprovat si realitzant més execucions, les solucions són més pròximes a la solució òptima.

En aquests resultats s'ha estudiat com són de bones les solucions proposades si fem diverses execucions de l'algoritme per als casos 1, 5, 10, 25, 50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100 i 1200 execucions. Per a fer-ho s'ha utilitzat el mateix mapa de 100 ciutats per realitzar les execucions de tots els casos, i s'ha comprovat l'aproximació que tenen a la solució proposada de 5000 execucions. Cada nombre d'execucions s'ha realitzat 3 cops per a comprovar els diferents fitness que pot retornar.

Per a 5000 execucions s'ha trobat que la millor solució proporciona un fitness de 6691, sent aquesta la millor solució proposada. I els resultats obtinguts amb la resta d'execucions han sigut els següents:

Nombre Execucions	Execució 1	Execució 2	Execució 3
1	8005	7713	7992
5	7780	7463	7665
10	7258	7591	7630
25	6998	7413	7494
50	7076	7285	7199
100	7012	7444	6989
200	6965	7374	7053
300	6834	6915	6926
400	6889	6884	7056
500	6922	7034	7060
600	6908	6786	6979
700	6726	6911	6781
800	6847	6736	6866
900	6788	6728	6881
1000	6817	6793	6777
1100	6761	6880	7000
1200	6908	6717	6750
5000	6691	6743	6819

Taula 9. Comparació solucions proposades amb diferent nombre d'execucions

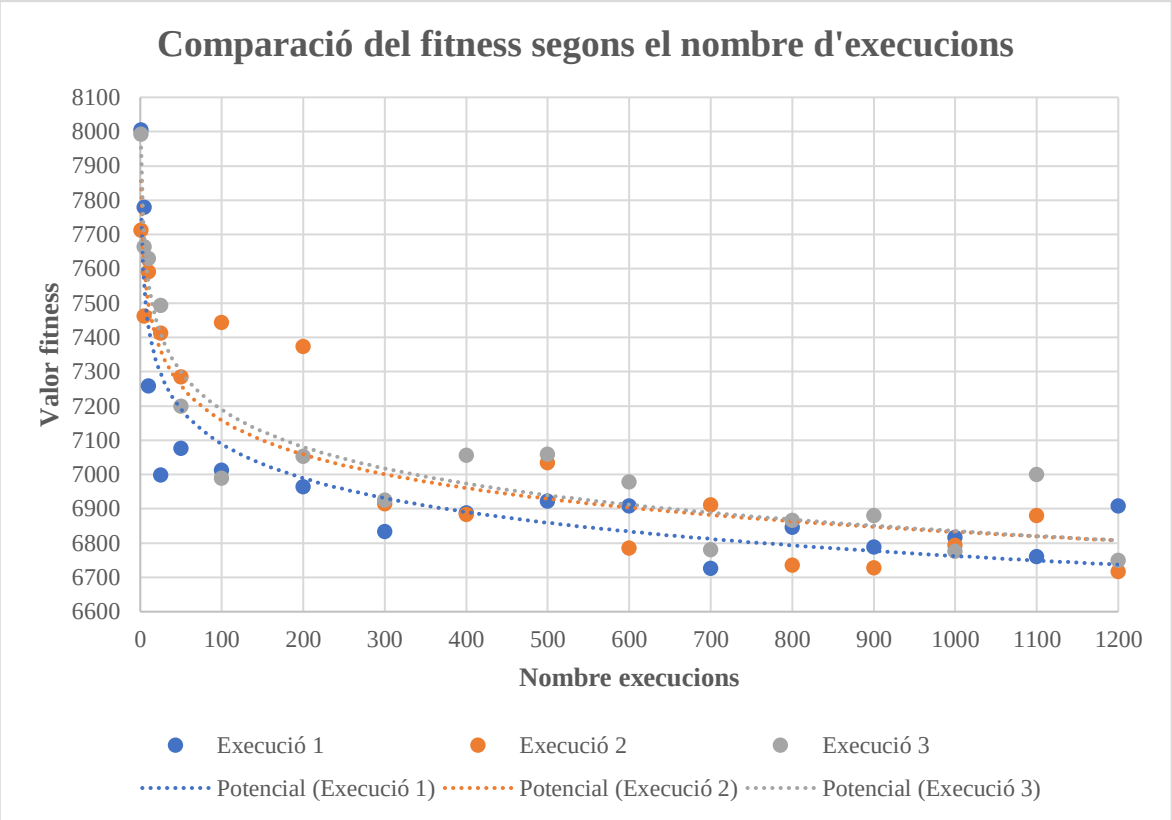


Figura 32. Gràfica comparació del fitness segons el nombre d'execucions

Amb els resultats obtinguts es pot veure com a mesura que s'augmenta el nombre d'execucions realitzades el valor del fitness s'aproxima cada vegada més a la solució general en comptes d'un mínim local. També es pot observar com en nombres d'execucions baixes la solució proposada és probable que estigui allunyada de la solució general òptima, i, que a l'augmentar el nombre d'execucions, ràpidament convergeix amb poques execucions, tal com es veu a la gràfica. Si es vol trobar la solució general, la capacitat de càlcul necessària augmenta de forma considerable, ja que a partir de les 200 execucions la millora de la solució ja no millora considerablement i ràpidament com amb poques execucions, i comença a reduir-se la millora fent-se cada vegada més difícil trobar una solució millor.

Per tant, es pot concloure que augmentant el nombre d'execucions realitzades és més probable obtenir la solució general o una molt pròxima amb facilitat i la capacitat de càlcul necessària serà major. Si s'augmenta el nombre de ciutats del problema, es necessitarà realitzar encara més execucions al dispersar-se més el rang de cerca del problema TSP, i per tant s'haurà d'incrementar la capacitat de càlcul.

5.3 Síntesis dels resultats obtinguts

A l'analitzar els resultats anteriors s'han pogut extreure una sèrie de conclusions sobre els beneficis i inconvenients d'utilitzar l'arquitectura Serverless per a l'execució d'algoritmes genètics, els quals ja han estat comentats una mica anteriorment.

El principal avantatge de la tecnologia Serverless és la **millora de rendiment** al paral·lelitzar-se de forma massiva un gran nombre d'execucions entre molts servidors, permeten executar les mateixes tasques en un **menor temps i de forma eficient**. A més a més, **l'elasticitat de l'escalabilitat** que ofereix és idònia per afrontar grans càrregues i disminucions de treball de forma puntual, ja que la reserva de recursos és ràpida i senzilla de fer.

A més a més, a l'augmentar el nombre d'execucions que es poden arribar a realitzar, permet trobar **solucions encara més bones en els algoritmes genètics** que si s'hagués d'executar en un àmbit local es necessitaria una gran capacitat de càlcul o s'hauria de reduir el nombre d'execucions a realitzar. Per tant, aporta una **millora considerable en la qualitat de les solucions** al facilitar moltes execucions simultànies sense grans temps d'execució.

Si a més a més es suma que amb el framework **Lithops** encara es **simplifica** més la feina de configurar els servidors Serverless per a **l'execució**, i que es pot canviar de proveïdor fàcilment, utilitzar la tecnologia Serverless és pràcticament com si s'estigués treballant de forma local però amb recursos il·limitats.

Tot i això, com totes les tecnologies, té els seus inconvenients que s'han pogut observar de forma senzilla. Principalment els **dos inconvenients** observats han sigut l'inici en fred i la inestabilitat en els temps d'execució.

Per a execucions on pràcticament les tasques a executar són molt breus i poca quantitat de tasques, hi ha l'inconvenient de **l'inici en fred**. Aquest inici en fred fa que els temps inicials d'arrancada dels servidors siguin molt més lents degut a que normalment els servidors han d'engegar-se pràcticament de zero per començar a executar. Per tant, es perd uns pocs segons en l'arrancada dels servidors quan es tracta amb execucions ràpides i poques quantitats, que en alguns casos resultaria molt més ràpid l'execució local.

La **inestabilitat en els temps d'execució** fa que no es pugui acabar de preveure el temps d'execució d'una tasca. Aquestes inestabilitats poden ser provocades per retards en la xarxa, denegacions de servei o fins i tot la congestió dels servidors en hores puntes.

Al comparar l'execució Serverless davant l'execució local es pot concloure que **la millora del Speedup que proporciona l'execució Serverless és realment considerable** a mesura que es necessita més capacitat de càlcul per a realitzar més execucions, les quals són necessàries si es vol obtenir la solució general de forma més senzilla i eficaç i així reduir el temps d'execució de l'algoritme.

A més a més, si es sumen els factors de **reducció de costos i manteniment**, i la **millor escalabilitat** que proporciona el model Serverless, aquest sobrepassa sens dubte el model local quan es treballa a baixa, mitja i gran escala, però no en una escala introductòria.

Per últim, també s'ha de tenir en compte que molts cops no és necessari la utilització de molts recursos per a l'execució Serverless, ja que si **s'adapten correctament el nombre de recursos** a les necessitats de càlcul del problema, o, fins i tot, sacrificant una mica de potència de càlcul necessari per a l'execució del problema, es poden **reduir els costos totals** de l'execució Serverless. Tot i així, els costos de creació i manteniment d'una infraestructura pròpia amb una gran capacitat de càlcul i emmagatzematge sempre seran molt majors als de lloguer de forma esporàdica d'aquests servidors.

6 Conclusions

Amb la creixent necessitat de recursos de computació elevats per a poder tractar amb la quantitat d'informació que existeix actualment, els sistemes Serverless poden resultar una solució atractiva per a molts usuaris. Al ser senzills d'utilitzar, amb un cost econòmic més atractiu per a clients esporàdics o que la seva necessitat de recursos no és constant, i la capacitat de computació que aporten gràcies a la seva eficiència i escalabilitat, són opcions realment interessants i a tenir en compte si es treballa amb el núvol. A més a més, gràcies a la gran quantitat de frameworks com Lithops i altres que cada vegada apareixen més, faciliten encara més la tasca d'interacció i treball amb el núvol abstraient les parts més complicades.

Després d'implementar una solució al problema TSP amb algoritmes genètics i estudiar el seu comportament tant en la seva execució local com amb l'execució amb la tecnologia Serverless, els sistemes Serverless són una opció que no descartaré si algun altre cop he de treballar amb tecnologies Cloud. A més a més, gràcies a aquest treball he pogut introduir-me en el gran món que és el Cloud, el qual està en constant desenvolupament i creixement i que qualsevol empresa o usuari amb necessitats més elevades pot aprofitar per millorar les seves idees, negocis o reduir costos.

Personalment ha resultat interessant fer un projecte més de recerca i no pas un d'implementació d'una pàgina web, perquè d'aquesta forma m'he pogut adonar de les dificultats de l'obtenció de resultats, errors que poden anar sorgint durant la implementació i anàlisis de resultats i extreure conclusions útils sobre un tema d'investigació concret. A més a més, m'ha permès fer-me una petita introducció a l'immens món del Cloud, que actualment és un dels aspectes tecnològics més demandats i innovadors del sector de la informàtica. M'hauria agradat encara aprofundir més en l'obtenció de resultats, però degut a la gran quantitat d'inconvenients que provoca la utilització de proves gratuïtes i altres no ha sigut possible aprofundir a un nivell de detall més gran que el que he pogut proporcionar. Tot i això, estic satisfet amb els resultats obtinguts i la feina elaborada.

7 Referències

[1] Pàgina Web https://en.wikipedia.org/wiki/Travelling_salesman_problem. [Explicació TSP]

[2] Article Web https://towardsdatascience.com/how-to-solve-the-traveling-salesman-problem-a-comparative-analysis-39056a916c9f. [Explicació TSP 2]

[3] Pàgina Web https://en.wikipedia.org/wiki/Genetic_algorithm. [Explicació algoritmes genètics]

[4] Article Web https://towardsdatascience.com/an-introduction-to-genetic-algorithms-c07a81032547. [Explicació algoritmes genètics 2]

[5] Pàgina Web https://en.wikipedia.org/wiki/Selection_(genetic_algorithm). [Explicació selecció]

[6] Pàgina Web https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm). [Explicació creuament]

[7] Pàgina Web https://en.wikipedia.org/wiki/Mutation_(genetic_algorithm). [Explicació Mutació]

[8] Pàgina Web https://www.ibm.com/Cloud/learn/serverless. [Explicació computació serverless]

[9] Pàgina Web https://www.ibm.com/Cloud/learn/faas. [Explicació funcions com a servei]

[10] Pàgina Web https://www.ibm.com/Cloud/learn/object-storage. [Explicació Object Storage]

[11] Repositori Github https://github.com/lithops-Cloud/lithops. [Repositori de Lithops-Cloud]

[12] Article Web https://towardsdatascience.com/parallel-and-distributed-genetic-algorithms-1ed2e76866e3 . [Algoritmes genètics de forma distribuïda]

[13] Pàgina web https://github.com/lithops-Cloud/lithops/blob/master/docs/design.md. [Disseny i flux d'execució de Lithops]

[14] Article Web https://www.hebergementwebs.com/tutorial-on-genetic-algorithms/genetic-algorithms-selection-of-parents. [Explicació algoritmes de selecció]

[15] Pàgina web https://en.wikipedia.org/wiki/Stochastic_universal_sampling. [Explicació Stochastic Universal sampling]

[16] Article web https://www.hindawi.com/journals/cin/2017/7430125/. [Explicació algoritmes creuament per a TSP]