Universitat de Girona
Escola Politècnica Superior

Grau en Enginyeria Informàtica

PROJECTE FINAL DE GRAU

Implementació i millora d'un algoritme de Time-Interval-Related-Pattern (TIRP) Mining incremental per ser executat en paral·lel i en C++

Autor:
Pau Calvera Moliner

Tutor:
Beatriz López Ibañez

MEMÒRIA

Convocatòria:
Juny 2023

Departament :
Enginyeria Elèctrica, Electrònica i Automàtica

Projecte: Projecte Final de Grau
Document: Memòria
Títol: Implementació i millora d'un algoritme de Time-Interval-Related-Pattern (TIRP) Mining incremental per ser executat en paral·lel i en C++
Autor: Pau Calvera Moliner
Data: Juny 2023

Estudi:
Grau en Enginyeria Informàtica
Universitat de Girona

Supervisor:
Beatriz López Ibañez
Universitat de Girona
Email: beatriz.lopez@udg.edu

Índex

Capítol 1

Introducció

El codi font del projecte es pot trobar en el següent repositori públic [1].

1.1 Motivació

Els algortimes de mineria de "Time-interval-related pattern (d'ara endavant TIRP) consisteixen en trobar patrons de relacions temporals tal com "A comença B". El seu ús pioner era trobar patrons de compres, però també pot tindre utilitat en altres camps, com per exemple per fer detecció i classificació en les seqüencies d'ADN, detectar atacs i/o buscar signatures dels virus, prediccions de preus dels estocs, etc. TIRP mining és una família de mètodes de "sequential patern mining"que permeten minar els patrons amb relacions totals o parcials en intervals temporals d'esdeveniments. El problema que presenten aquesta classe d'algoritmes és que tenen un cost computacional elevat i és per això que es fa de manera incremental: en aquest cas al tractar-se d'una estructura de dades en arbre, es computa per nivells.

En l'article [2], s'introdueix un nou algoritme per minar TIRPS anomenat "vertTIRP", que combina una representació eficient d'aquests patrons utilitzant les propietats de transitivitat que tenen, amb una estratègia d'aparellament que ordena les relacions temporals per així accelerar el procés de mineria. A més presenta una definició robusta de les relacions temporals que elimina les ambigüitats entre relacions fent ús d'un valor "èpsilon". En l'article esmentat es pot trobar un enllaç del repositori de l'algoritme implementat en python.

1.2 Objectius

Com he explicat en la secció anterior, aquesta classe d'algoritmes tenen un cost computacional elevat. L'objectiu del projecte és implementar l'algoritme "vertTIRP"en C++ per disminuir el temps de càlcul i comparar els resultats amb el "vertTIRP"original. El llenguatge C++ és conegut per ser ràpid i eficient, a més disposa de punters que millorarien significativament l'eficiència i diverses opcions de multithreading.

Capítol 2

Estudi de viabilitat

A continuació analitzaré la viabilitat tecnològica, és a dir els recursos de maquinari i de programari requerits, i la viabilitat econòmica, on es tindran en compte els recursos tecnològics i humans implicats en el projecte.

2.1 Recursos tecnològics

Per desenvolupar l'algoritme *vertTIRP* he necessitat els següents recursos tecnològics:

- **Un Ordinador** amb connexió a internet per dur a terme tot el desenvolupament del projecte. El sistema operatiu pot ser tant *Windows* com *Linux*.
- **Clion**[3] IDE per a c/c++ desenvolupat per *Jetbrains*. He escollit aquest perquè és el que m'agrada més i tinc la llicència gratuïta de la universitat, però es pot fer servir qualsevol altre.
- **Pycharm**[4] IDE per a python desenvolupat també per *Jetbrains*. Igual que amb el *Clion* es pot fer servir qualsevol altre.
- **Bitbucket**[5] Servei d'allotjament basat en web per a projectes basats en *git*. L'algoritme original està penjat en aquesta plataforma.
- **Github**[6] Per publicar el projecte i també per poder treballar amb el codi en cas d'utilitzar un ordinador diferent.
- **Overleaf**[7] Editor de LaTeX en línia per redactar la memòria del projecte.
- **Libreoffice Calc**[8] Full de càlcul desenvolupat per *Libreoffice* per gestionar els resultats i fer gràfics.
- **Draw.io**[9] Plugin de *Google drive* per fer diagrames d'aplicació.
- **Valgrind**[10] Eina de depuració de per trobar fuites de memòria.
- **GanttProject**[11] Eina per elaborar diagrames de Gantt [12].
- **G++**[13] Compilador per a C++.
- **Python 3.10.11**[14] Llenguatge de programació Python.

- **OpenMp**[15] Interfície de programació multiprocés.
- **MySQL**[16] Sistema de gestió de bases de dades relacionals que utilitza el llenguatge SQL.
- **HeidiSQL**[17] Interfície de programació multiprocés per a varis sistemes de gestió de bases de dades (entre els quals es troba el *MySQL*).

2.2 Viabilitat econòmica

El pressupost dels recursos tecnològics ascendeix a un total de 1.078,00€ (veure Taula 2.1). Cal destacar que la major part d'aquest cost està destinat a l'adquisició de l'ordinador. Pel que fa a les llicències del Clion i el Pycharm, les he obtingut gratuïtament com a estudiant. A més, existeixen moltes alternatives sense cap despesa addicional. Les altres eines i plataformes que he utilitzat són gratuïtes, i no suposen cap cost extra.

Concepte	Preu
PC	600.00€
Clion	229.00€
Pycharm	249.00€
Bitbucket	0.00€
Github	0.00€
Overleaf	0.00€
Libreoffice Calc	0.00€
Draw.io	0.00€
Valgrind	0.00€
GanttProject	0.00€
TOTAL	1.078,00€

TAULA 2.1: Pressupost recursos tecnològics.

Concepte	Temps	Cost/Hora	Total
Programador junior	375 hores	10€/hora	3.750.00€
TOTAL			3.750.00€

TAULA 2.2: Pressupost recursos humans.

El pressupost de recursos humans, només contempla un programador junior que durà a terme tot el projecte. Aquesta persona dedicarà un total de temps d'unes 375 hores amb un sou de 10€/hora, el que suposa un cost total de 3.750.00€ (veure Taula 2.2).

Capítol 3

Metodologia

Durant la realització d'un Projecte de fi de grau has d'escollir quina metodologia utilitzaràs per dur a terme el projecte i així assolir els objectius que prèviament he descrit (veure 1.2).

Al llarg del grau hem vist diferents metodologies amb les quals es poden treballar, però quan vaig decidir de que tractaria el treball, la metodologia que millor s'adaptava a aquest projecte i a la meva manera de treballar és la metodologia *àgil*[18], concretament el tipus *SCRUM*[19].

La metodologia *àgil*[18] és un tipus de metodologia que permet desenvolupar projectes de manera flexible i adaptativa, i augmenta la capacitat de respondre als canvis que es presenten durant el desenvolupament de projecte, és a dir solucionar problemes de forma àgil i realitzar correccions o millores de manera contínua, en lloc d'esperar fins al final del projecte per fer els ajustaments i correccions pertinents. Encara que això m'ha suposat un avantatge també m'ha suposat que el projecte requerís més temps del que havia planificat en un principi (veure capítol 4). Un altre element que es va veure afavorit per la meva elecció va ser la qualitat del producte final, ja que com ja he dit aquest tipus de metodologia permet fer revisions contínues.

Hi ha diversos tipus d'implementacions de la *àgils*:

- **Scrum[19]:** és una metodologia àgil que se centra en la realització de tasques incrementals per a lliurar un producte funcional al final de cada sprint. Cada sprint és un període de temps determinat.
- **Kanban[20]:** aquest tipus de metodologia se centra en la visualització del flux de treball, la limitació del treball i l'optimització del flux per reduir els temps d'espera i millorar la productivitat.
- **Lean[21]:** es centra en la millora contínua del producte i del procés, així com en la reducció dels residus i l'eliminació dels residus i activitats que no afegeixen valor al producte final.
- **Extreme Programming (XP)[22]:** es centra en la realització de tasques iteratives i incrementals, així com en la millora contínua del producte i el procés, posa un fort èmfasi la realització de proves automatitzades.

Jo personalment per la classe de projecte i per la meva manera de treballar vaig considerar que la modalitat més adequada és la tipus *Scrum* [19]. Vaig dividir el meu

projecte en diferents *sprints* que inicialment havien de tenir una durada d'una setmana, però al final van acabar durant un període d'una a tres setmanes com s'explica al capítol 4.

Capítol 4

Planificació

Abans de començar el projecte, vaig fer una planificació del desenvolupament estimant el cost en termes de temps que tindria cada tasca, és a dir cada *sprint* (capítol 3). A partir d'aquesta planificació vaig fer un diagrama de *Gantt* [12] (veure figura 4.1) amb l'estimació inicial del temps que em portaria cada tasca.

En el procés de desenvolupament, em vaig trobar amb diversos reptes que van afectar la planificació del projecte. Un dels principals reptes va ser l'anàlisi i comprensió del codi original, especialment en les parts més complexes. A l'inici del projecte, com es mostra en la planificació inicial (veure Figura 4.1), l'anàlisi del codi va ser identificat com una tasca prèvia als *sprints* (capítol 3). No obstant això, en haver planificat sense saber exactament com funciona cada classe, no vaig calcular massa bé el temps real que hi acabaria invertint.

Quan vaig fer l'estimació inicial, vaig preveure que la part de programació requeriria aproximadament 150 hores de treball. Malgrat això, durant el transcurs del projecte, em vaig adonar que el temps real dedicat a aquesta tasca va ser el doble del previst, aproximadament 375 hores (veure Figura 4.2).

També, vaig veure que l'anàlisi del codi és un procés continu durant tot el projecte. Inicialment, va consistir a entendre el funcionament general del codi original. No obstant això, a mesura que vaig anar programant funcions i classes, va ser necessari revisar de manera específica com funcionen en detall. Aquesta revisió i consulta constant del codi existent va ser una part integral del procés de traducció.

Per resumir, l'anàlisi i comprensió del codi original s'ha mostrat com un procés recurrent i en evolució durant tot el desenvolupament del projecte, el qual requereix una atenció continuada i un ajustament adequat de la planificació inicial.

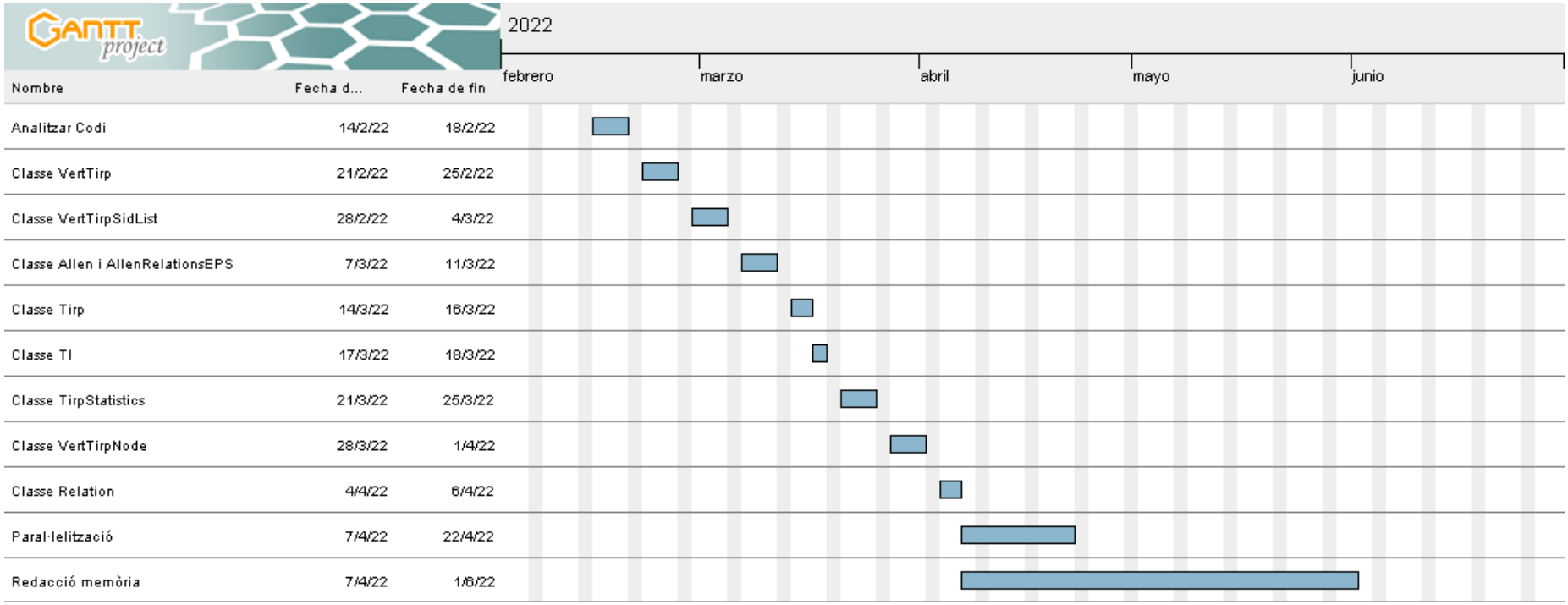


FIGURA 4.1: Diagrama de *Gantt* de la planificació inicial del projecte. Cada recuadre del cronograma simbolitza una setmana. El transcurs de la planificació va de la tercera setmana de ferbrer, a la primera de juny.

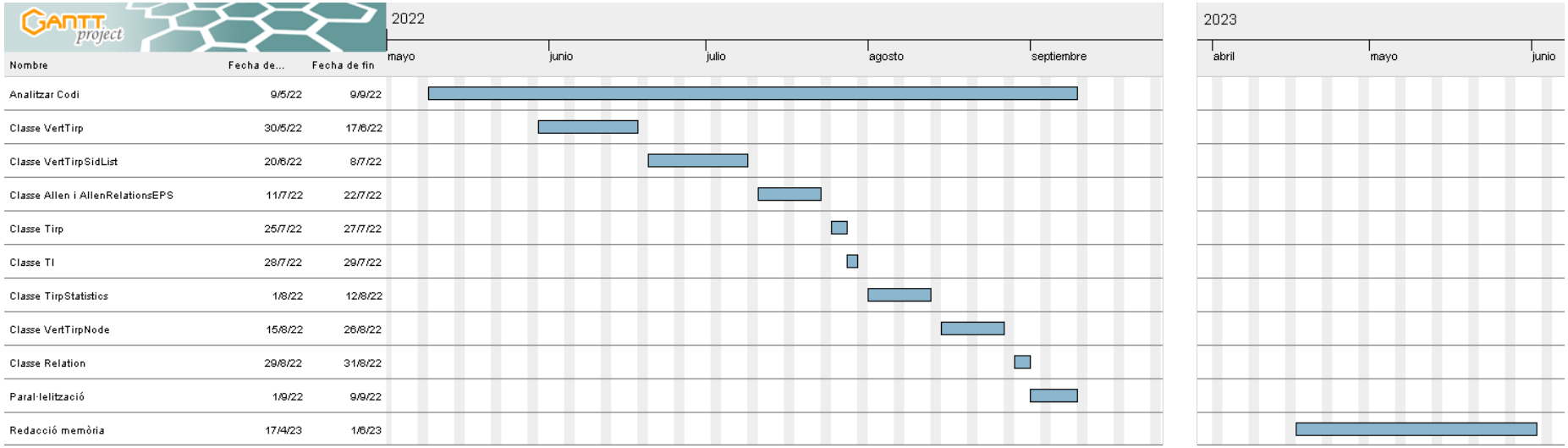


FIGURA 4.2: Diagrama de *Gantt* de la planificació real del projecte. Cada recuadre del cronograma simbolitza una setmana. El transcurs real ha anat de la segona setmana de maig del 2022, a la primera de juny del 2023.

Capítol 5

Marc de treball i conceptes previs

5.1 Marc de treball

Els algoritmes de mineria de *TIRPs* (Time-Interval-Related Pattern) són una categoria específica d'algoritmes utilitzats per descobrir patrons d'intervals de temps. Els TIRPs són patrons que capturen relacions entre intervals de temps, com ara "A comença B"o "A se solapa amb B". Els algoritmes de mineria de *TIRPs* L'objectiu és extreure aquests patrons de la forma més eficient possible.

Aquests utilitzen diverses tècniques per identificar *TIRPs*, incloent representacions de dades, propietats de transitivitat (vegeu seccions 9.1 i 9.2) i estratègies per ordenar i provar relacions temporals. L'objectiu és accelerar el procés de mineria i reduir la complexitat computacional que aquesta classe d'algoritmes comporten.

En l'article [2] se'ns presenta l'algoritme *vertTIRP*, que combina una eficient representació d'aquests patrons, fent ús de transitivitat temporal (seccions 9.1 i 9.2) i una estratègia d'emparellament (secció 9.8.1) que canvia l'ordre de càlcul de les relacions temporals per tal d'agilitzar el procés. A més, afegeix un tractament d'ambigüitat a l'hora de calcular les relacions temporals entre intervals amb un valor *èpsilon* i dues restriccions addicionals (secció 5.2).

5.2 Conceptes previs dels els algoritmes TIRP

Quan parlem d'algoritmes de tipus *TIRP*, és important conèixer alguns conceptes que s'expliquen en detall a l'article [2]. En aquesta secció, llistaré els més rellevants.

- **Ti (Time interval)** $Ti = \langle sym, s, e \rangle$ Consta d'un simbol que el representa, un temps d'inici i de fi.
- **IS (Seqüència d'intervals)** $IS = \langle Ti^1, Ti^2, ..., Ti^n \rangle$
- **Quasi igual ($=^\epsilon$)** Dos intervals són quasi iguals si es compleix $|t^i - t^j| < \epsilon$.
- **Precedeix ($<^\epsilon$)** Un interval precedeix un altre si es compleix $|t^j - t^i| > \epsilon$.
- **LSTIS (Lexicographical Symbolic Time Interval Sequence)** $LSTIS = \langle Ti^1, Ti^2, ..., Ti^n \rangle$ És una seqüencia de *Ti* ordenada de menor a major.
- **TIRP (Time Interval Related Pattern)** $TIRP = \langle SeqSym, s, \alpha \rangle$ On SeqSym és

una seqüencia de símbols i α les relacions entre cada una de les parelles entre símbols. Per exemple $\langle ABC, sbo, \alpha \rangle$ significa ("A starts B","A before C","B overlaps C").

- **S-TIRP (Time Interval Related Pattern)** $S - TIRP = \{TIRP^1, TIRP^2, ..., TIRP^n\}$ On tots els TIRP que conté tenen la mateixa seqüencia de símbols. p.e. $BC = \{\langle BC, o \rangle, \langle BC, m \rangle\}$
- **Suport vertical d'un TIRP** En quantes seqüències es compleix un TIRP.
- **Suport horitzontal d'un TIRP** El nombre de vegades que un TIRP coincideix en una seqüència.
- **Mitjana de suports horitzontals d'un TIRP** La mitjana de suport horitzontal de cada seqüencia.
- **Mitjana de duració d'un TIRP** la mitjana de duració a partir de les instàncies del TIRP que podem trobar en totes les seqüencies.

5.2.1 Relacions temporals

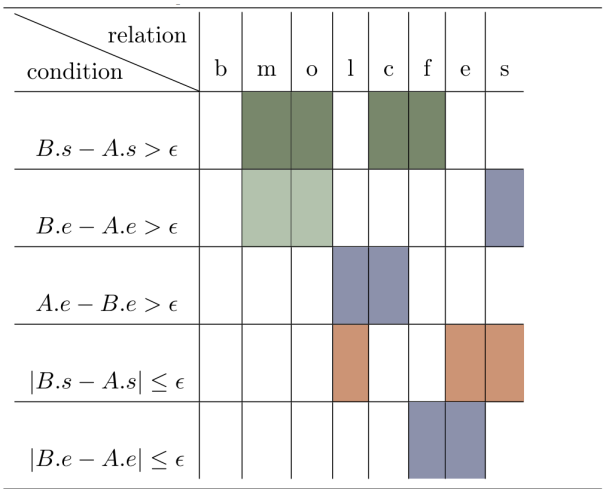
Donats dos intervals de temps(TI) format per A i B, podem identificar 8 tipus de relacions temporals (vegeu la taula 5.1). Aquestes relacions es calculen a partir del temps d'inici (ex. $A.s$) i de fi (ex. $A.e$) dels dos intervals:

$$\text{Before} \rightarrow b$$
$$\text{Meets} \rightarrow m$$
$$\text{Overlaps} \rightarrow o$$
$$\text{Contains} \rightarrow c$$
$$\text{Finish by} \rightarrow f$$
$$\text{Equal} \rightarrow e$$
$$\text{Starts} \rightarrow s$$
$$\text{Left contain} \rightarrow l$$

5.3 Conceptes previs específics de l'algoritme vertTirp

En aquesta secció es llisten els conceptes més importants de l'algoritme *vertTIRP*.

- **Restricció de durada mínima ($C_{min\_duration}$)** La duració mínima que tindrà cada *TIRP*, els que no arribin a aquesta duració es descartaran.
- **Restricció de bretxa mínima (C_{min_gap})** Els intervals de temps que tinguin una relació $t^i <^\epsilon t^j$ hauran d'estar separats per almenys C_{min_gap} unitats de temps.
- **Intersecció d'intervals temporals** Donades dues relacions temporals α i β i les condicions que les defineixen $\lambda(\alpha)$ i $\lambda(\beta)$, la interseccio $\alpha \cap \beta$ és el conjunt de totes les condicions $\lambda(\alpha)$ que també pertanyen a $\lambda(\beta)$, és a dir $\lambda(\alpha) \cap \lambda(\beta)$
- **Grup d'intervals temporals** Dues relacions temporals α i β estan en el mateix grup si comparteixen una sola condició lògica.
- **Subgrup d'intervals temporals** Dues relacions temporals α i β estan en el mateix subgrup si comparteixen al menys dues condicions lògiques i existèix una tercera relació temporal γ que comparteix una d'aquestes dos condicions.



TAULA 5.1: Taula que mostra les condicions que requereix cada relació temporal (vegeu secció 5.2.1). Obtinguda de l'article [2].

Capítol 6

Requisits del sistema

6.1 Requisits funcionals

Els requisits funcionals de *vertTIRP*, és a dir els paràmetres d'entrada que necessita l'algoritme, són els següents:

- **Suport vertical (numèric entre 0 i 1):** El suport vertical mínim que han de tenir els *TIRPs* computats.
- **Epsilon (numèric):** Valor utilitzat per tractar L'Ambigüitat al calcular relacions temporals.
- **Estratègia d'aparellament (llista):** Ordre de testeig de les relacions temporals.
- **Transitivitat (boleà):** Determina si s'utilitzaran taules de transitivitat per accelerar el temps d'execució.
- C_{min_gap}**:** Restricció de bretxa mínima al calcular relacions temporals.
- C_{max_gap}**:** Restricció de bretxa màxima al calcular relacions temporals.
- $C_{min\_duration}$**:** Duració mínima que ha de tenir un interval de temps per a ser considerat.
- $C_{max\_duration}$**:** Duració màxima que pot tenir un *TIRP* per a ser considerat.
- C_{Dummy} **(boleà):** Determina si es farà una execució *dummy*, és a dir sense fer ús d'estratègia d'aparellament.
- **Intervals de temps per seqüència (llista):** Llista amb els intervals de temps per a cada seqüència.
- **Seqüències (llista):** Llista amb els noms de les seqüencies a minar.

6.2 Requisits no funcionals

Els requisits no funcionals de *vertTIRP* en c++ són els següents:

- **Mantenir la integritat:** En tractar-se d'una traducció, l'algoritme és exactament el mateix i, per tant, el funcionament ha de ser el mateix. Les estructures de dades, tot i que puguin ser diferents (per motius de rendiment), han de representar exactament les mateixes dades i funcionar de la mateixa manera. Aquesta integritat és important sobretot en paral·lelitzar, ja que existeixen certes estratègies per podar l'arbre a generar que tenen en compte el funcionament seqüencial de l'algoritme.

- **Rendiment:** El principal motiu per traduïr el codi a c++, és millorar el temps d'execució, ja que el c++ és molt més ràpid que el python i és el llenguatge que se sol utilitzar per algoritmes d'aquest estil.

- **Llegibilitat del codi:** El fet d'utilitzar c++ fa que la lectura del codi sigui més complexa (sobretot a l'haver de definir tipus per cada variable i l'ús de punters). Per tant, he considerat un requisit rellevant mantenir el màxim possible la simplicitat del codi, estructurant-lo de la mateixa forma, fent servir els mateixos noms per les variables i amb comentaris en el mateix codi.

Capítol 7

Estudis i decisions

7.1 Paral·lelització

Una CPU està dissenyada per realitzar una àmplia varietat de tasques, com ara processament de dades, càlculs matemàtics, control del sistema operatiu, gestió de memòria (en poca latència), i altres tasques relacionades amb el funcionament global del sistema. En una CPU, cada nucli està optimitzat per executar tasques en seqüencial, on s'executa un seguit d'instruccions una darrera de l'altra, per aquesta complexitat, conté una quantitat reduïda de nuclis.

D'altra banda, una GPU està dissenyada específicament per al processament paral·lel massiu. Té una arquitectura més paral·lela i conté molts més nuclis, tot i que més senzills (operacions aritmètiques més simples). És molt útil per tasques com operacions amb matrius, renderització 3D, simulacions científiques, etc.

En un principi la meva intenció era paral·lelitzar a través de GPU (veure 7.1.1), però me'n vaig adonar que per com funciona l'algoritme era més factible utilitzar la CPU (veure 7.1.2). A continuació s'explica detalladament.

7.1.1 Primera opció: OpenCl

FIGURA 7.1: Logo d'opencl.

Les dues *APIs* més importants per programar amb GPU són *CUDA* [23] i *OpenCl* (figura 7.1[24]). Com que *CUDA* és una propietària de *Nvidia* i només es pot utilitzar en les GPUs de la seva marca, em vaig decantar per *OpenCl* que es pot executar en qualsevol dispositiu.

Com que la GPU treballa amb càlculs computacionals senzills, L'única part de l'algoritme que vaig trobar que es podia paral·lelitzar són les funcions per calcular les

relacions temporals entre dos intervals de temps de la secció 9.11.1. Aquestes funcions es criden principalment des del mètode *calc_rel* (veure 9.11.2) que consisteix en varis bucles que van calculant les relacions fins que es troba una de vàlida. Vaig pensar que podia pre-calcular totes les relacions en paral·lel amb *OpenCl* i guardarles en una taula, de manera que el bucle simplement hi accedeixi sense perdre temps esperant que es facin els càlculs. No obstant em vaig adonar, de que no valia massa la pena per tres motius:

- Els rangs dels bucles no són prou grans. És a dir, al tenir una estratègia d'aparellament i taules de transitivitat, ja es limita molt el nombre de relacions per calcular. A més utilitzar una GPU surt a compte quan hi ha molta informació a paral·lelitzar ja que el fet d'haver d'enviar totes les dades a la memòria a través del bus *pci-e* no és instantani i pot crear overhead. El que vull dir amb això és que en cas de que hi hagi una millora de temps sospito que no pot ser molt gran.

- Pot ser poc eficient energèticament, ja que si pre-calcules tots els càlculs que es poden fer en un bucle però llavors en la primera iteració ja troba una relació vàlida. La resta s'han computat per res.

- Amb la classe *Chrono* (verue 9.9) vaig calcular el temps d'execució total de cada funció i, tot i que hi està un percentatge del temps total important (més o menys un 30%) en cas de que millores de temps no suposaria una millora massa important en el temps total d'execució.

Per tant, vaig decidir descartar l'opció d'utilitzar una GPU i em vaig decantar per fer-ho amb CPU, ja que la resta d'element paral·lelitzables de l'algoritme consisteixen en processament de dades.

7.1.2 OpenMP

FIGURA 7.2: Logo d'openmp.

Finalment m'he decidit per OpenMp, (figura 7.2) és una *API* per programar en paral·lel en C++. És senzilla d'utilitzar i el compilador g++ (el que he utilitzat jo) l'importa amb el flag "-fopenmp", per afegir les llibreries simplement s'ha d'afegir *include <omp.h>* en el fitxer en el que ho vulguem utilitzar.

L'objectiu és paral·lelitzar la funció de mineria de dades, és a dir la funció *dfs_pruning()* (veure 9.4). Aquesta funció crida la funció *join()* (veure 9.6) que al seu temps crida a *update_tirp_attrs*. Amb la classe *Chrono* he analitzat el temps i he vist que pràcticament la totalitat del temps d'execució és aquesta ultima funció, ja que en realitat *dfs_pruning* crida en bucle a *join* i aquesta es un conjunt de bucles dins de bucles que criden a *update_tirp_attrs()*. Per tant es pot paral·lelitzar qualsevol *for()* d'una d'aquestes tres funcions.
Vaig provar amb tots els *for()* per separat per veure quin donava millors resultats, però em vaig donar compte de que el rendiment empitjorava. No només això sino

que, com més exterior sigui el *for* i com més threads utilitzes pitjor resultats obtenia. Això és degut al overthread, vaig veure que com més variables compartides entre *threads* i més zones crítiques (zones que només es poden executar en un thread per evitar problemes d'integritat de dades) pitjor és el rendiment. Així dons, vaig decidir paral·lelitzar el bucle de la funció *update_tirp_attrs*(codi 7.1) que agafa un vector de *TIRPs* (veure 9.3) i per cada posició genera un *TIRP* extès.

LISTING 7.1: Paral·lelització amb OpenMp

```
#pragma omp parallel for
        default(none)
        private(index)
        shared(extended_tirps,
               tirps_to_extend,
               tirps_to_extend_size,
               f_ti_0,eps,
               min_gap,
               max_gap,
               max_duration,
               mine_last_equal,
               ps)
        num_threads(omp_get_num_threads())
        schedule(dynamic)
for ( index = 0 ; index < tirps_to_extend_size ; index++)
    extended_tirps[index] = tirps_to_extend[index]
    ->extend_with(f_ti_0,
                  eps,
                  min_gap,
                  max_gap,
                  max_duration,
                  mine_last_equal,
                  ps);
```

Aquest for es el millor per paral·lelitzar perquè el mètode *extend_with()* de la classe *TIRP* no modifica l'instancia sinó que en crea una de nova, per tant no hi han regions crítiques perquè no hi ha cap dada que pugui ser modificada per dos *threads* de forma concurrent, ja que cada un està processant un element diferent de la taula *tirps_to_extend*.
La directiva *pragma omp parallel for* indica que el *for* que ve a continuació s'executarà en paral·lel, llavors es defineixen una serie clàusules per configurar de quina forma es farà. La clausula *private()* diu que la variable index és privada a cada *thread*, és a dir cada un té la seva pròpia, així dons la clàusula *shared()* indica les vaiables compartides, és a dir que qualsevol *thread* hi pot accedir i si un la modifica queda canviada per tots els altres. La clàusula *num_threads* indica en quants *threads* s'executarà el bucle i *schedule(dynamic)* significa que no hi ha un nombre fixe d'iteracions per *thread*, sinó que s'aniran assignant a mida que aquests les vagin processant. Aquesta última clàusula és important perquè a vegades hi han posicions de la taula que tarden més temps a processar-se que d'altres i així es reparteix millor la feina entre tots els *threads*.

Per tal de minimitzar al màxim l'overhead, he fet que els atributs de la classe *TIRP* (veure 9.3) siguin de tipus bàsics, he canviat els vectors de la llibreria *std* per taules clàssiques de C. No obstant al tenir tantes variables compartides en cada *thread* és inevitable que hi hagi overthread, i com s'ensenya al capítol 10 la paral·lelització només surt rentable en alguns casos.

7.2 Eines de traducció automàtiques

En un primer moment vaig pensar en utilitzar alguna eina de traducció automàtica per agilitzar el procés sobretot en les parts més monòtones i repetitives. Alguns exemples d'aquestes eines són: *ChatGPT*[25], *Github Copilot*[26] o el *CodeConvert*[27]. Però al final vaig decidir no utilitzar-les ja que tenen una serie d'inconvenients que fan que no es puguin complir tots els requisits de projecte (veure 6.2):

- **Compatibilitat limitada:** Certs llenguatges permeten fer coses que són complicades d'implementar en altres. Per exemple, en *Python*, pots tenir una llista que contingui taules o caràcters sense problemes, mentre que en C++ hauràs de crear una classe que consideri aquestes dues possibilitats (9.8.1).

- **Optimització:** A l'hora de traduir el codi d'un llenguatge a un altre, és essencial aprofitar les propietats específiques del nou llenguatge per optimitzar certes parts del codi. Aquesta tasca requereix un anàlisi detallat del funcionament de l'algoritme, que no pot ser realitzat per una eina de traducció automàtica. Tal com s'explica a la secció 8.2, l'optimització és una etapa crucial en el procés de traducció, en especial un algoritme de mineria de *TIRPs*.

- **Semàntica:** Els llenguatges de programació poden diferir significativament en la seva semàntica, i una traducció automàtica pot no capturar adequadament aquestes diferències. Això pot ser un problema especialment quan es tradueixen llenguatges molt diferents entre si, com ara llenguatges que operen sota diferents paradigmes. En el cas d'aquest projecte crec que no hagués estat un problema, ja que la semàntica dels dos llenguatges no es difereix en excés però considero que és important comentar-ho.

En conclusió, tractant-se d'un codi complex en un àmbit de recerca, aquestes eines no són una bona opció.

Capítol 8

Anàlisi i disseny del sistema

8.1 Passos principals

A continuació explico els passos principals en l'anàlisi del sistema (figura 8.1). L'anàlisi comença per comprendre l'algoritme original, identificar els diferents components del codi i les estructures de dades que conté. A partir d'aquest anàlisi i amb enginyeria inversa, he obtingut un Diagrama de classes extès (figura 8.2). També ha estat important identificar les dependències i llibreries, ja que són diferents en els dos llenguatges. Finalment, i a partir d'aquest anàlisi prèvi, he dut a terme la recodificació del codi.

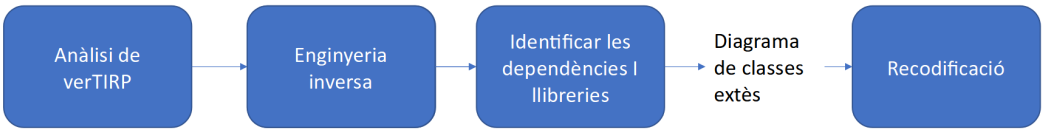


FIGURA 8.1: Diagrama del passos principals per dur a terme la traducció.

8.2 Traducció de codi

A continuació enumeraré els punts que m'han semblat més importants a l'hora de dur a terme una traducció informàtica, concretament en la d'aquest projecte:

- **Analitzar en profunditat el codi original**. És important conèixer bé tant com funciona el mateix codi, com el llenguatge de programació. En el meu cas, l'algoritme original està escrit en python, un llenguatge que hem tocat relativament poc al llarg de la carrera, i per tant he estat un repte més en el procés de comprendre el codi. Per altra banda, m'ha estat molt útil el *el debugger* sobretot per veure com es gestionen les dades entre estructures i com funciona internament cada cosa.

- **Bon coneixement del llenguatge a utilitzar**. El propòsit del projecte és traduir el codi a C++, en aquest llenguatge ja tenia un coneixement previ robust perquè és un dels que més hem fet servir al llarg de la carrera, sobretot els primers anys. En les assignatures de *Metodologia i tecnologia de la programació* com a *Estructures de dades i algorítmica* s'ha treballat molt tant l'eficiència com el bon ús de la memòria, dues coses que han estat importants en el projecte.

- **Començar per una base**. En el meu cas, vaig decidir començar creant totes les classes buides i els seus atributs, així tindria una estructura per on començar a programar. Com és lògic, la majoria d'atributs no estaven ben declarats, ja que en molts de casos no sabia quins tipus assignar-los-hi, per tant, vaig deixar molts *TODO* pel codi que més endavant acabaria canviant. Llavors a partir d'aquesta base, i seguint la metodologia escollida (vegeu capítol 3), vaig anar implementant classe per classe dividint la feina per funcions. També és útil començar per les classes petites que serveixen més com a contenidors de dades i no contenen subclasses, i llavors passar a les més complexes.

- **Proves d'execució** És important fer proves d'execució i comprovar que encaixin amb les mateixes proves en l'algoritme original. En el capítol 10 es troben documentades aquestes proves.

- **Optimització** Una vegada es té traduït el codi principal, es poden fer optimitzacions per adaptar-lo millor al llenguatge. Amb C++, la utilització dels punters de C++, i l'ús de memòria estàtica i dinàmica pot marcar la diferència en el temps d'execució.

8.3 Diagrama de classes

A partir del codi *python* original, he fet un anàlisi exhaustiu i per enginyeria inversa he obtingut el diagrama de classes (veure 8.2).

A diferència de l'algoritme original, he afegit les classes *Relation*(9.8) i *AllenRelationsEPS*(9.11.1) així com els fitxers *Utils*(9.1) i *Global*(9.10), el motiu està explicat en les seves respectives seccions.

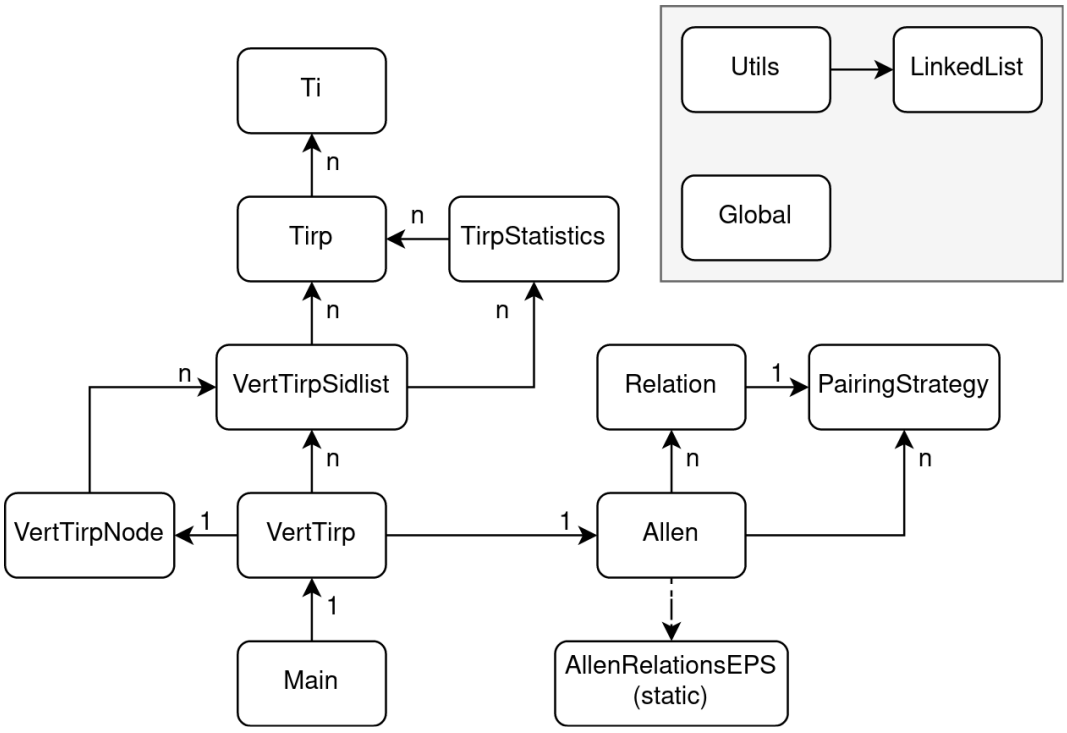


FIGURA 8.2: Diagrama de classes de l'algoritme en C++.

8.4 Anàlisi de l'algoritme original

En l'article [2] s'explica detalladament el funcionament de l'algoritme, en aquest capítol destaco les coses més importants per entendre com funciona.

El procés de mineria es fa a partir d'una llista d'intervals de temps (veure exemple figura 8.3) repartits entre vàries seqüencies. Aquests intervals s'identifiquen amb un símbol (p.e. A,B,C) i poden estar repetits encara que sigui amb diferents temps. Per exemple en la figura 8.3 l'interval C dura de 11:00 a 13:00 en la primera seqüència , i també de 10:00 a 13:00 i de 14:00 a 15:00 en la tercera seqüència.

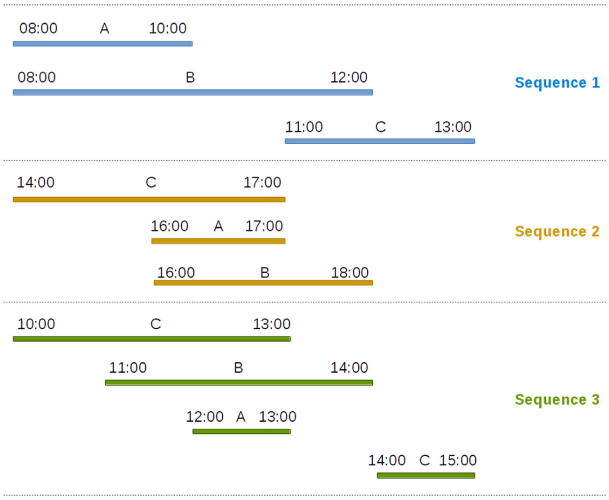


FIGURA 8.3: Exemple d'escenari amb seqüències temporals

L'objectiu de l'algoritme és, a partir de les dades d'entrada (figura 8.3) generar un arbre amb tots els possibles patrons (*TIRPs*) (figura 8.4). Cada node de l'arbre es tracta d'un *S-TIRP* que representa una combinació d'intervals i conté una serie de *TIRPS* amb les relacions entre aquests intervals. El primer nivell està format per *S-TIRPS* de mida 1, a partir d'aquest nivell es van generant els següents nivells de forma recursiva i per cada nivell la mida de la seqüencia de símbols dels *S-TIRPS* augmenta en 1. Aquest arbre pot créixer de manera exponencial fins que o bé s'acaben totes les combinacions o fins que s'arriba a *TIRPS* amb un suport vertical o horitzontal límit especificat prèviament (per podar la cerca i reduir el temps d'execució).

8.5 Adaptació del codi a C++

L'objectiu principal d'adaptar l'algoritme a C++ és l'eficiència. Al tenir una complexitat exponencial interessa que el temps d'execució sigui el mínim possible. Per aquest motiu he dedicat bastant esforç en optimitzar el màxim possible el codi, passant i retornant objectes per referència sempre que es pugui, fent ús de punters, utilitzant les estructures de dades més adients per cada situació, etc. A més, per tal de que el codi sigui entenedor, vaig decidir programar-ho des de zero utilitzant la mateixa estructura, noms de variables i les mateixes classes (figura 8.2). No obstant, per la nomenclatura del llenguatge pot haver-hi alguna part que sigui més complicada d'entendre, sobretot en la classe *VertTirpSidList*(9.6) que fa servir molts iteradors i comprovacions d'elements en mapes.

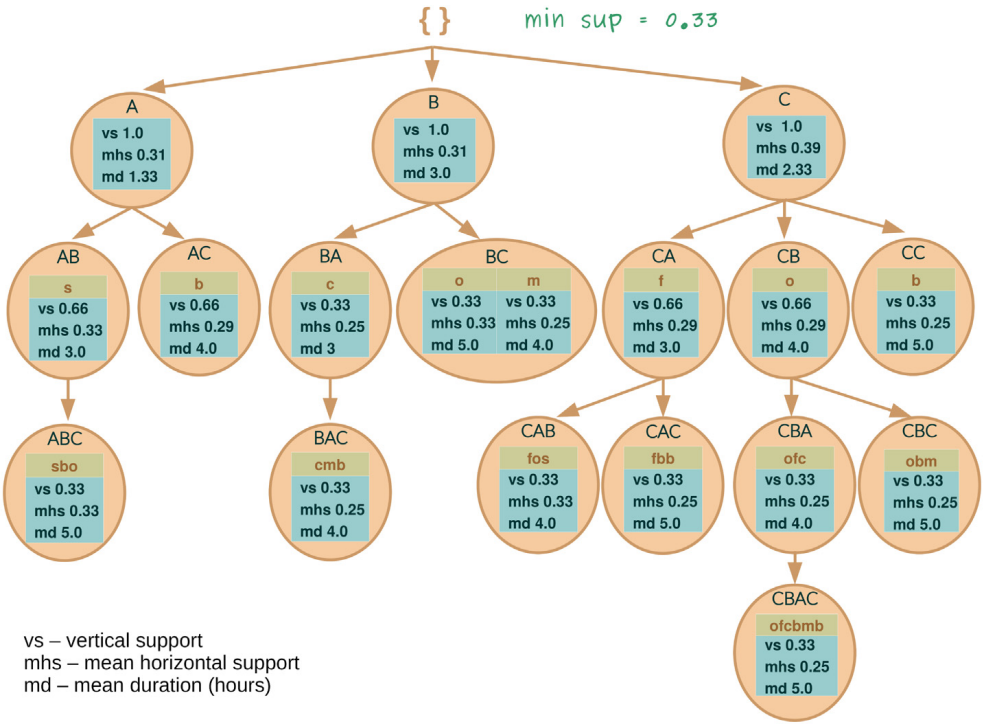


FIGURA 8.4: Arbre generat a partir de l'exemple 8.3

Com que utilitzar punters amb C++, té el risc de que hi hagin fuges de memòria si no s'utilitzen correctament, he utilitzat l'eina *Valgrind* [10], que executa el codi analitzant la gestió de la memòria dinàmica per determinar si hi han fuges i en cas de que hi siguin mostrar-les.

8.6 Tractament dels datasets

He utilitzat tres datasets per a les proves, com s'explica a la secció 2.1: *has* [28], *har* [29] i *mav* [30].
En l'article [2], es detallen els datasets utilitzats, però no detalla exactament com s'han agrupat les dades per formar els intervals de temps i les seqüències. Això m'ha obligat a improvisar i formatejar-les de la manera que he considerat més adequada. Per aquest motiu, el nombre d'intervals de temps no coincideix exactament amb l'article, tot i que s'hi aproxima.

Per processar les dades, donada la seva gran quantitat d'informació, les he inserit en una base de dades *MySQL* [16], amb una taula per a cada dataset. A continuació, mitjançant una consulta *SELECT* (annexos B.1, B.2 i B.3), els he donat format i exportat com a fitxers *CSV*, per tal que puguin ser llegits per l'algoritme.

Així, he pogut manipular i adaptar les dades per a l'ús posterior en l'estudi.

id	BodyAcc	GravityAcc	BodyGyro	subject	Activity	time
1	b	a	b	1	STANDING	2010-01-01 10:00:00
2	b	a	b	1	STANDING	2010-01-01 10:00:02
3	b	a	b	1	STANDING	2010-01-01 10:00:05
4	b	a	b	1	STANDING	2010-01-01 10:00:07
5	b	a	b	1	STANDING	2010-01-01 10:00:10
6	b	a	b	1	STANDING	2010-01-01 10:00:12
7	b	a	b	1	STANDING	2010-01-01 10:00:15
8	b	a	b	1	STANDING	2010-01-01 10:00:17
9	b	a	b	1	STANDING	2010-01-01 10:00:20
10	b	a	b	1	STANDING	2010-01-01 10:00:23
11	b	a	b	1	STANDING	2010-01-01 10:00:25
12	b	a	b	1	STANDING	2010-01-01 10:00:28
13	b	a	b	1	STANDING	2010-01-01 10:00:30
14	b	a	b	1	STANDING	2010-01-01 10:00:33
15	b	a	b	1	STANDING	2010-01-01 10:00:35
16	b	a	b	1	STANDING	2010-01-01 10:00:38
17	b	a	b	1	STANDING	2010-01-01 10:00:40
18	b	a	b	1	STANDING	2010-01-01 10:00:43
19	b	a	b	1	STANDING	2010-01-01 10:00:46
20	b	a	b	1	STANDING	2010-01-01 10:00:48
21	b	a	b	1	STANDING	2010-01-01 10:00:51
22	b	a	b	1	STANDING	2010-01-01 10:00:53
23	b	a	b	1	STANDING	2010-01-01 10:00:56
24	b	a	b	1	STANDING	2010-01-01 10:00:58
25	b	a	b	1	STANDING	2010-01-01 10:01:01
26	b	a	b	1	STANDING	2010-01-01 10:01:04
27	b	a	b	1	STANDING	2010-01-01 10:01:06
28	b	b	b	1	SITTING	2010-01-01 10:01:09

FIGURA 8.5: Base de dades amb el dataset *har* original.

sid	start_time	end_time	value
1	1,262,336,599	1,262,336,776	abaWALKING
1	1,262,336,776	1,262,337,040	abaWALKING_DOWNSTAIRS
1	1,262,337,040	1,262,337,180	abaWALKING
1	1,262,337,180	1,262,337,206	abaWALKING_UPSTAIRS
1	1,262,336,620	1,262,336,720	abcWALKING
1	1,262,336,720	1,262,336,784	abcWALKING_DOWNSTAIRS
1	1,262,336,784	1,262,337,034	abcWALKING_UPSTAIRS
1	1,262,337,034	1,262,337,157	abcWALKING
1	1,262,337,157	1,262,337,175	abcWALKING_DOWNSTAIRS
1	1,262,337,175	1,262,337,242	abcWALKING_UPSTAIRS
1	1,262,336,568	1,262,337,006	accLAYING
1	1,262,336,400	1,262,336,471	babSTANDING
1	1,262,336,471	1,262,336,850	babSITTING
1	1,262,336,850	1,262,336,917	babSTANDING
1	1,262,336,917	1,262,337,004	babSITTING
1	1,262,336,476	1,262,336,476	bacSITTING
1	1,262,336,469	1,262,336,481	bbbSITTING
1	1,262,336,479	1,262,336,479	bbcSITTING
1	1,262,336,530	1,262,337,032	bcbLAYING
1	1,262,336,576	1,262,337,009	bccLAYING
1	1,262,336,602	1,262,336,722	cbaWALKING
1	1,262,336,722	1,262,337,042	cbaWALKING_DOWNSTAIRS
1	1,262,337,042	1,262,337,162	cbaWALKING
1	1,262,337,162	1,262,337,285	cbaWALKING_DOWNSTAIRS
1	1,262,336,730	1,262,337,265	cbcWALKING_DOWNSTAIRS
2	1,262,355,600	1,262,355,612	abaWALKING_UPSTAIRS
2	1,262,355,423	1,262,355,561	abcWALKING
2	1,262,355,561	1,262,355,802	abcWALKING_UPSTAIRS
2	1,262,355,802	1,262,355,935	abcWALKING

FIGURA 8.6: Resultat d'aplicar la comanda al dataset *har* per poder ser llegit per l'algortime.

Capítol 9

Implementació i proves

9.1 Fitxer Útils

L'algoritme original conté un fitxer anomenat *ti2lstis.ty* dedicat a la lectura de les dades del dataset, jo ho he decidit crear el fitxer *Utils.cpp*. Aquest fitxer conté, funcions d'utilitat que he anat necessitant al llarg del desenvolupament del projecte, funcions referents a la lectura de fitxers, i l'implementació de la classe *LinkedList*.

LISTING 9.1: Funcions d'utilitat

```
1 string utils_unifyStrings(vector<string> &seq_str_strings);
2 string utils_unifyChars(string &seq_chars);
3 dif_time_type truncate(dif_time_type t);
```

En el codi 9.1 es poden veure les funcions d'utilitat que he afegit de més al projecte. Són bastant senzilles però les he hagut d'implementar ja que hi han algunes coses que python permet però C++ no. Per exemple les funcions de *unifyStrings* i *unifyChars* bàsicament agafen un vector de strings i caràcters respectivament i retornen un string amb la llista separada per comes i entre claudàtor.
La funció *truncate()*, simula la funció *ttu()* de l'algoritme original tot i que l'implementació és diferent. El motiu és que tal com s'explica a la secció 8.4, l'algoritme original té tres modes de temps, tant el primer com el tercer són numèrics, però el segon és la classe *Timedelta*. Com que a l'hora de fer càlculs es necessita un valor, la funció *ttu()* crida el mètode *total_seconds()* del objecte el qual dona el temps en segons. En l'algoritme en C++ aquest mode de temps no hi és, ja que es tracta d'una llibreria de python, i per tant ho fa tot amb valors numèrics, el problema és que la funció *total_seconds()* de python retorna zero si el nombre de segons és més petit que 1e-6. Per tal que els resultats siguin els mateixos en els dos algoritmes, la funció *truncate()* arrodoneix el nombre a zero quan es dona la condició.

9.1.1 Lectura de fitxers

La lectura de fitxers funciona bastant semblant que en l'algoritme original. Una de les desavantatges de treballar amb C++ és l'absència de llibreries que faciliten les coses, al no poder utilitzar el paquet *pandas*, he fet una funció que llegeix el fitxer, guarda cada linia en l'struct Csv_line (codi 9.2), i les agrupa per el nom de la seqüència: *map<string, vector<Csv_line> >*.

LISTING 9.2: Struct amb un interval de temps

```
struct Csv_line {
    string start_time;
    string end_time;
    vector<string> values;
};
```

Una vegada es tenen totes les línies guardades es creen els intervals de temps (classe Ti 9.2) i es guarden en llistes encadenades (classe LinkedList 9.1.2). El resultat final de la lectura són dos vectors de mateixa mida 9.3, un amb els noms de les seqüències, i l'altre amb les llistes d'intervals de temps de cada sequència.

LISTING 9.3: Dades de lectura processades

```
vector<LinkedList> list_of_ti_users;
vector<string> list_of_users;
```

9.1.2 Classe LinkedList

Tot i que la llibreria [31] conté estructures de dades doblement encadenades, he hagut de crear una de pròpia perquè per fer una inserció ordenada, ho fa en funció de l'ultim element inserit i l'ordre final no és el mateix que agafant una llista doblement encadenada normal i ordenant-la.

Durant el procés d'anàlisi del codi, em vaig donar compte que en cap moment del codi es modifiquen les instàncies de les classes *Ti* 9.2 ni *Tirp* 9.3, i com que cada vegada que es guarden en alguna estructura de dades es creen copies, vaig decidir gestionar-les amb punters. En la llista encadenada es guarden totes les instàncies de la classe *Ti* 9.2, i com que en cap moment de l'execució se'n crearan de noves ni s'eliminaran, he fet servir punters normals (en comptes de smart pointers [32]) ja que s'eliminaran en el destructor de la classe LinkedList en el moment de finalitzar l'algoritme.

9.2 Classe TI

La classe Ti (codi 9.4) és bastant senzilla, representa un interval de temps: simbol, temps d'inici i temps de fi. Aquests atributs s'assignen en el constructor i una vegada definida l'instancia no es modifica. Per tant només hi han definits els *getters*, els operadors de comparació per poder ordenar i l'operador *ostream* per imprimir.

LISTING 9.4: Header de la classe Ti

```
class TI {
public:
    TI(string sym, time_type start, time_type end);
    time_type get_start()const;
    time_type get_end()const;
    string& get_sym();

    friend ostream &operator<<(ostream &os, const TI &ti);
    bool operator==(const TI &rhs) const;
    bool operator!=(const TI &rhs) const;

    bool operator<(const TI &rhs) const;

private:
    string sym;
    time_type start;
    time_type end;
};
```

9.3 Classe Tirp

Tal com he explicat a la secció 9.1.2 les instàncies de les classes *Ti* 9.2 i *Tirp* 9.3 les he tractat amb punters, però a diferencia de les *Ti*, en les instàncies *Tirp* he utilitzat *shared_ptr*[32] ja que no les tinc totes guardades en una estructura de dades que després pugui esborrar de manera segura, sinó que es van creant i copiant en el procés de mineria, així si s'esborra una estructura de dades amb instàncies, només s'eliminaran les que siguin úniques, és a dir si només existeix un punter que les contingui.

LISTING 9.5: Atributs de la classe Tirp

```
1     TI** ti;
2     time_type first;
3     time_type max_last;
4     char* r;
5     int ti_size;
6     int r_size;
```

Una instància de *tirp* (Time-interval-related Pattern) conté els atributs especificats en el codi 9.5: una taula amb els punters d'instàncies *Ti* ordenades i una taula de caràcters que representen les relacions entre aquestes instàncies. Els atributs *first* i *max_last* són el temps que comença el primer interval i el temps que acaba l'ultim respectivament, d'aquesta manera tenim el temps d'inici i final del *tirp* així com a durada total d'aquest.

LISTING 9.6: Mètode *Mètode per afegir un Ti*

```
1 pair<shared_ptr<TIRP>,unsigned> extend_with(TI* s_ti,
2                                  eps_type eps,
3                                  time_type min_gap,
4                                  time_type max_gap,
5                                  time_type max_duration,
6                                  bool mine_last_equal,
7                                  Allen &allen)const;
```

La funció més important és la que serveix per estendre el *Tirp* amb una instància de *Ti* 9.6, però com que l'objecte no es pot modificar, se'n crea un de nou on la taula *ti* té una posició més, i i la taula de relacions té *ti_size* relacions més, ja que per cada interval de temps ja existent s'ha de calcular la relació amb el que hem afegit. Un cop creat l'objecte es retorna un *shared_ptr* i un número que indica l'estat.

9.4 Classe VertTirp

LISTING 9.7: Header de la classe VertTirp (sense els arguments de les funcions)

```
class VertTIRP{
public:
    VertTIRP( . . . );
    int mine_patterns( . . . );
    void print_patterns( . . . );
private:
    Allen allen;
    map<string,unsigned> events_per_sequence;
    map<string,VertTirpSidList> vertical_db;
    vector<string> f1;
    VertTirpNode tree;

    string out_file;
    support_type min_sup_rel;
    float min_confidence;
    int min_gap;
    time_type max_gap;
    time_type min_duration;
    time_type max_duration;
    int max_length;
    int min_length;
    eps_type eps;
    int tirp_count;
    support_type min_sup;
    int time_mode;

    bool same_variable( . . . );
    void dfs_pruning( . . . );
    void to_vertical( . . . );
};
```

La classe vertTirp (codi 9.7), encapsula tot l'algoritme i conté tots els mètodes i atributs que aquest requereix. Té tres funcions publiques:

- **El constructor**, crea tota la classe a partir dels arguments.
- **mine_patterns()**, s'encarrega d'executar l'algoritme en si, amb les dades de lectura 9.1.1, primer crida a *to_vertical()* que inicialitza el mapa *vertical_db* amb instàncies de *vertTirpSidlist* de llargada 1. Tot seguit, per cada una d'aquestes instàncies crida la funció recursiva *dfs_pruning* que anirà generant l'arbre de combinacions.
- **print_patterns()**, imprimeix els resultats, és a dir l'arbre de l'atribut *tree* en el fitxer de sortida (o per consola) ja sigui en preordre o per nivells.

9.5 Classe TirpStatistics

LISTING 9.8: Header de la classe TIRPstatistics

```
class TIRPstatistics {
public:
    . . .
private:
    map<string,map<unsigned,vector<shared_ptr<TIRP>>>>
    sequence_events_tirps_dict;
    unsigned sum_ver_supp;
    map<string,unsigned> sum_hor_per_seq;
    string last_modified;
    vector<time_type> sum_mean_duration;
    vector<unsigned> n_instances_per_seq;
    vector<time_type> mean_duration;
};
```

Aquesta classe (codi 9.8) no hi he fet cap canvi important, representa les dades per calcular els valors estadístics d'un *TIRP*, és a dir, suport vertical, horitzontal, mitjana de duracions,etc. L'atribut *sequence_events_tirps_dict* conté els altres *TIRPs* de l'*S-TIRP* al que pertany.

9.6 Classe VertTirpSidlist

LISTING 9.9: Header de la classe VertTirpSidlist

```
class VertTirpSidList {
public:
    VertTirpSidList();
    VertTirpSidList join( . . . );
    unsigned update_tirp_attrs( . . . );
private:
vector<string> seq_str;
map<string,map<unsigned,vector<shared_ptr<TIRP>>>>
    definitive_ones_indices_dict;
map<string,shared_ptr<TIRPstatistics>> definitive_discovered_tirp_dict;
map<string,shared_ptr<TIRPstatistics>> temp_discovered_tirp_dict;
unsigned n_sequences;
unsigned support;
};
```

La classe *VertTirpSidList*(codi 9.9) representa un *S-TIRP* (veure 8.4), és la classe més densa i complexa ja que conté les funcions *join()* i *update_tirp_attrs()*. L'atribut *definitive_discovered_tirp_dict* conté instàncies de *TIRPstatistics*(veure 9.5) indexats per strings que representen les relacions, i l'atribut *definitive_ones_indices_dict* és un mapa indexat per seqüències on cada ítem és un mapa de vectors de *TIRPs*.

La funció *join()* crea una instància *VertTirpSidlist* resultant de fusionar l'actual amb una segona que se li passa per paràmetre. Això ho fa creant una copia de l'original i li va afegint TIRPs amb la funció

9.7 Classe VertTirpNode

LISTING 9.10: Header de la classe VertTirpNode

```
class VertTirpNode {
public:
    struct Node {
        vector<shared_ptr<Node>> child_nodes;
        string patt;
        unsigned pat_len;
        VertTirpSidList sidlist;
        bool is_root;
    };
    VertTirpNode();
    VertTirpNode( . . . );
    void add_child( . . . );
    void print_tree( . . . );
    static void print_tree_dfs( . . . );
    static void print_tree_bfs( . . . );

private:
    shared_ptr<Node> node;
};
```

Aquesta classe (codi 9.10) es tracta d'un arbre n-ari d'instàncies *VertTirpSidList*. Es fa servir per representar el resultat de l'algoritme com en la figura d'exemple 8.4. Està implementat amb punters i cada node conté l'string *patt* que són els simbols dels intervals de temps, la mida, *sidlist* que est tracta del S-TIRP i un vector dels nodes fills.

El mètode *print_tree()* reb el nom d'un fitxer i un booleà que indica si s'ha d'imprimir l'arbre en *dfs* o *bfs*. Llavors es crida a una de les dues funcions estàtiques segons el valor boleà i li passa l'atribut node (ja que les funcions són estàtiques), i el fitxer *ostream*[31] obert. En cas de que el fitxer no es pugui obrir, es passa *cout* que també és de tipus *ostream* de manera que l'arbre s'imprimeixi en consola.

9.8 Classe Relation

LISTING 9.11: Header de la classe Relation

```
class Relation {
public:
    Relation(string s);
    Relation(pair< . . . > p);
    bool isString()const;
    string getString()const;
    pair<shared_ptr< . . . >,shared_ptr< . . . >>& getPairingStrategy();
    unsigned size()const;
private:
    bool isS;
    string s;
    pair<shared_ptr<PairingStrategy>,shared_ptr<vector<string>>> p;
};
```

El mètode *get_pairing_strategy* de la classe *AllenRelationsEPS* (veure 9.11.1) retorna dues taules que representen l'estratègia d'aparellament, la segona és una taula d'strings normal, però la primera és una taula de taules on els valors d'aquestes poden ser un caràcter o una tercera taula de caràcters. Python permet que una taula pugui tindre valors de diferents tipus però C++ no, per això he creat la classe *PairingStrategy* (veure 9.8.1) per representar-la.
A més, quan es guarden les estratègies d'aparellament al atribut *sorted_trans_table* de la classe *Allen* (veure 9.11.2) en alguna ocasió en comptes de guardar les dues taules es guarda simplement un caràcter. Per solucionar aquest problema he encapsulat l'estratègia d'aparellament a la classe *Relation* (codi 9.11) on els atributs són una parella amb les dues taules (la primera representada amb la classe *PairingStrategy* i la segona amb un *vector*), el caràcter, i un valor booleà que diu quin dels dos atributs és el vàlid.

9.8.1 Classe PairingStrategy

L'implementació de la classe *PairingStrategy* (codi 9.12) és similar a la classe *Relation* (codi 9.11), es tracta d'un vector de vectors de tipus *Node*. L'struct *Node* conté un caràcter, un vector de caràcters i un booleà que determina quin dels dos atributs anteriors és el vàlid.

LISTING 9.12: Header de la classe PairingStrategy

```
struct Node {
    char cont;
    vector<char> l;

    bool dif;  //FALSE-valor    TRUE-llista valors
    Node(char c, bool i);
};

class PairingStrategy {
public:
    PairingStrategy();
    PairingStrategy(const PairingStrategy &o);
    void append(char c);
    void append2(char c);
    void appendAdd(char c);
    void add(char c);
    void add2(char c);
    bool empty()const;
    vector<vector<Node>> & get_ps();

private:
    vector<vector<Node>> ps; // representaci  de  l arbre
};
```

9.9 Classe Chrono

LISTING 9.13: Atributs globals

```
class Chrono {
public:
    Chrono();
    void start(string name);
    void stop(string name);
    void print()const;

    map<string,double> t;
    map<string,chrono::high_resolution_clock> taux;
};
```

He creat aquesta petita classe (codi 9.13) per medir el temps d'execució de diverses parts del programa. Això ha set bastant útil per analitzar en quines parts fixar-me a l'hora d'optimitzar. El funcionament és molt senzill, el mètode *start* guarda el temps en el moment de cridar-lo i el mètode stop calcula la diferència i la suma a la que ja tingui guardada (es va acumulant). L'argument *name* és per poder guardar més d'un temps ja que els va guardant en un *map* indexat per strings.

9.10 Fitxer Globals

LISTING 9.14: Atributs globals

```
typedef double time_type;
typedef double dif_time_type;
typedef float support_type;
typedef float eps_type;

using namespace std;

const time_type MAXGAP = 3155695200;
const string NONE = "N";
const string EMPTY = "";
const unsigned UNITS_NUMBER[6] = {1,60,60 * 60,60 * 60 * 24, 60 * 60 * 24
    * 7, 60 * 60 * 24 * 365};
const string UNITS_STRING[6] = {"seconds", "minutes", "hours", "days", "
    weeks", "years"};
const shared_ptr<vector<string>> grArr_nullPtr = NULL;
const shared_ptr<PairingStrategy> relsArr_nullPtr = NULL;
```

El fitxer *Globals* (veure 9.14) l'he creat per guardar-hi les variables que no depenen d'una classe en concret i es consulten des de diversos llocs. També hi he definit certs tipus perquè quan vaig començar a programar, no tenia molt clar quins tipus donar a algunes variables numèriques i així és molt més fàcil de modificar-los a posteriori.

9.11 Classe Allen i AllenRelationsEPS

L'algoritme original conté un fitxer anomenat *allen_relationsEPS.py* on estan definides les taules de transitivitat 9.1 i 9.2, les funcions que s'utilitzen per calcular les relacions temporals entre intervals de temps, i la classe *Allen*, que juntament amb aquestes funcions i les taules de transitivitat, calcula i assigna les relacions temporals. Com que en c++ no es pot accedir a variables fora d'una classe, he implementat la classe *AllenRelationsEPS* (veure 9.15) que conté aquestes estructures de dades i mètodes de forma estàtica. Una altra opció era declarar-ho tot de en el fitxer *Global* (secció 9.10) però per fer-ho més entenedor ho he agrupat tot en una sola classe, ja que només s'accedeix a aquestes dades des de *Allen*.

9.11.1 AllenRelationsEPS

LISTING 9.15: Header de la classe AllenRelationsEPS

```
class AllenRelationsEPS {
public:
    static pair<shared_ptr<PairingStrategy>,shared_ptr<vector<string>>>
    get_pairing_strategy(string str_rels);

    static unordered_map<string,Relation> trans_table_0;
    static unordered_map<string,Relation> trans_table;
    static unordered_map<char,int*> predefined_rels;

    static unordered_map<char,function<...>> ind_func_dict;
    static unordered_map<string,function<...>> cond_dict;
    static unordered_map<char,function<...>> rel_func_dict;

private:
    //--------------------- AUX FUNC ---------------------
    static int* before_ind( . . . );
    .
    .
    .
    static bool true_cond( . . . );
    //------------------ DUMMY AUX FUNC ------------------
    static int*  before( . . . );
    .
    .
    .
    static int* left_contains( . . . );
};
```

El canvi més important respecte l'algoritme original està en les funcions que calculen cada relació temporal (línies 15-26 del codi 9.15), aquestes funcions retornen dues dades, un caràcter, que indica la possible relació (en cas de que no sigui possible '\0') i l'estat, un nombre que pot tenir tres valors; 1 si la relació és correcte, 2 si no es compleix la restricció *min_gap* i 3 si no es compleix la de *max_gap*.
Em vaig fixar en que aquestes funcions sempre tornen una de 12 parelles possibles, i com que es criden moltes vegades, per no crear dades innecessàries he declarat prèviament les dotze variables de forma estàtica, i he fet que les funcions retornin un punter a qualsevol d'aquestes variables. Cada una d'aquestes és una taula de dues posicions (la segona dada és un caràcter però es pot tractar com un nombre i el càsting es fa automàtic) i com que les he declarat de forma estàtica i es generen en temps de compilació no hi ha problemes de possible pèrdua de memòria. Aquestes funcions no és criden directament, sinó que es fa a partir dels atributs *ind_func_dict*,*cond_dict* i *rel_func_dict* (veure 9.15). Aquests atributs són taules de hash (de tipus *unordered_map*) que retornen una funció o una altra a partir d'un caràcter. El motiu d'aquesta implementació és que com s'explica en l'article de l'algoritme original [2], el procés de mineria és fa amb una estratègia d'aparellament que ordena les relacions temporals i les representa amb caràcters.

Els atributs *trans_table_0* i *trans_table* (codi 9.15), representen taules de transitivitat (9.1 i 9.2 respectivament). Aquestes taules redueixen les possibiles relacions entre dos intervals, donat que sabem la relacio entre A i B, i la relació entre B i C, podem limitar el domini de relacions entre A i C.
El motiu per utilitzar un *unordered_map* és que respecte el *map* el temps d'accés sol ser més ràpid (O(1) en el millor dels casos).

A r1 BB r2 C	b	c	o	m	s	f	e
b "before"	b	b	b	b	b	b	b
c "contains"	b c f m o	c	c f o	c f o	c f o	c	c
o "overlaps"	b	b c f m o	b m o	b	o	b m o	o
m "meets"	b	b	b	b	m	b	m
s "starts"	b	b c f m o	b m o	b	s	b m o	s
f "finished-by"	b	c	o	m	o	f	f
e "equal"	b	c	o	m	s	f	e

TAULA 9.1: Taula de transitivitat d'Allen. [33]

A r1 BB r2 C	b	c	o	m	s	f	e	l
b "before"	b	b	b	b	b	b	b	b
c "contains"	b c f m o	c	c f o	c f o	c f o	c f	c f	c
o "overlaps"	b	b c f m o	b m o	b m	m o	b f m o	m o	c f o
m "meets"	b	b m	b m	b m	b m	b m	b m	b m
s "starts"	b	b c f m o	b m o	b m	m s	b m o	e m o s	c f l m o
f "finished-by"	b m	c f	f m o	b m o	f m o	c f m o	c f l m o	c f
e "equal"	b m	c f	f m o	b e m o	e o s	c f m	c e f o	c f l
l "left contain"	b c f m o	c	c f o	c m o	c e l o	c f	c e f l	c

TAULA 9.2: Taula de transitivitat per *eps* > 0. [2]

9.11.2 Allen

LISTING 9.16: Header de la classe Allen

```
class Allen {
public:
    Allen( . . . );
    int* calc_rel( . . . )const;
    int* assign_rel( . . . )const;
    Relation& get_possible_rels( . . . );
    bool get_trans()const;
private:
    string calc_sort;
    bool dummy_calc;
    bool trans;
    eps_type eps;

    shared_ptr<PairingStrategy> rels_arr;
    shared_ptr<vector<string>> gr_arr;
    unordered_map<string,Relation> sorted_trans_table;
};
```

La classe Allen de l'algoritme en python és una interfície que té dues implementacions, una per fer calculs en *dummy* i l'altre sense. Com que el C++ no permet les interfícies, he fet una sola classe amb l'atribut booleà *dummy_calc* (veure codi 9.16) i quan es crida un mètode, es fan les instruccions que toquen segons aquesta variable. Els atributs *rels_arr* i *gr_arr* són punters perquè la funció *get_pairing_strategy* (veure 9.15) que els inicialitza retorna punters per així no fer una copia dels objectes al moment de retornar-los.

Capítol 10

Implantació i resultats

Per calcular els resultats, he realitzat proves d'execució exactament iguals en els dos algoritmes, funcionant sota els mateixos paràmetres i i donant el mateix resultat. Per comprovar que donen el mateix resultat he he comprovat els fitxers de sortida que es generen.

Cal destacar, que en datasets grans, on es fan molts de càlculs amb decimals, els resultats poden diferir lleugerament entre els dos algoritmes per diferències d'aproximació numèrica. No obstant la càrrega de treball de l'algoritme és la mateixa i els resultats són pràcticament els mateixos.

10.1 Entorn de proves

- **Sistema operatiu:** Linux Ubuntu 22.04
- **CPU:** Ryzen 5600x 6-Core 4.2GHz
- **Memòria:** 32Gb RAM 3600Hz
- **Emmagatzematge:** SSD PCIe 500 Gb
- **Placa base:** Gigabyte b550 Aorus elite v2

10.2 Temps d'execució en seqüencial

Els resultats d'adaptar l'algoritme a C++, són positius. Hi ha una diferència bastant notable en el temps d'execució del codi, les figures 10.1, 10.2 i 10.3 mostren una comparació del temps entre els dos algoritmes amb el dataset *ASL*. Com es pot observar la diferència de temps entre els dos és semblant, no obstant la major diferència és en l'execució sense transitivitat i en *dummy* (figura 10.3) és a dir sense fer us de d'estratègies d'aparellament, on la millora de velocitat és d'un 11.814%. Cal destacar pero, que el que millors resultats en quant a temps és l'execució amb transitivitat i estratègia d'aparellament (figura 10.2). Segons els valors calculats en la millora de rendiment el temps d'execució en C++ representa més o menys un 1% del temps que tarda l'algoritme en python sota les mateixes condicions.

També he realitzat comparacions de temps utilitzant els conjunts de dades MAV (figura 10.5) i HAR (figura 10.4), obtenint resultats similars.

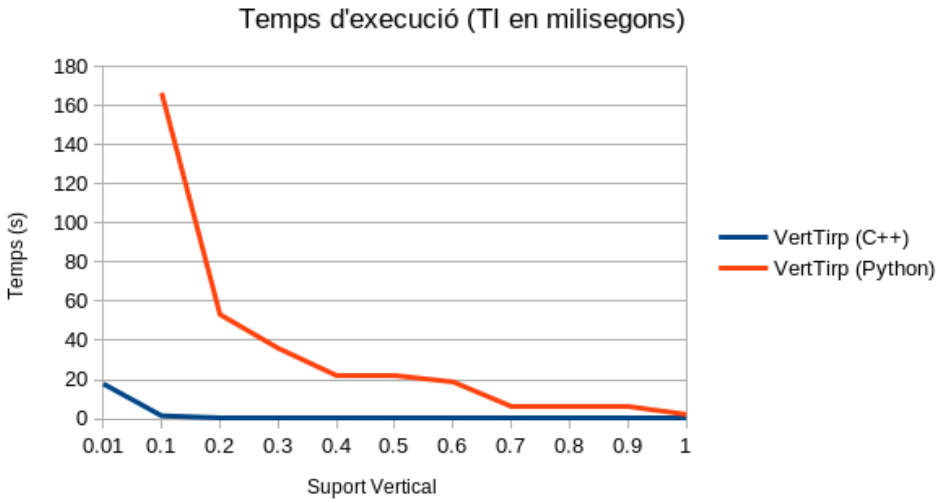


FIGURA 10.1: Temps d'execució del dataset *ASL* amb intervals de temps en milisegons. (per veure les dades exactes consultar l'annex A.1)

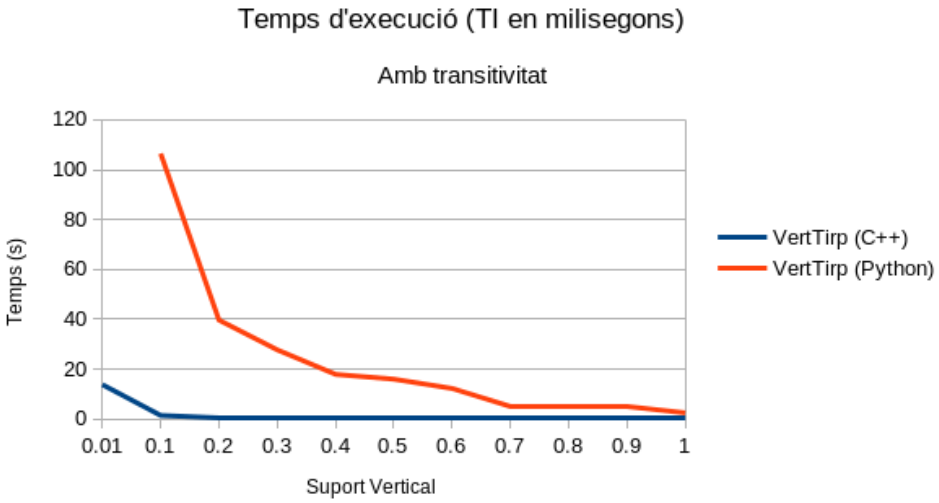


FIGURA 10.2: Temps d'execució del dataset *ASL* amb intervals de temps en milisegons i transitivitat. (per veure les dades exactes consultar l'annex A.2)

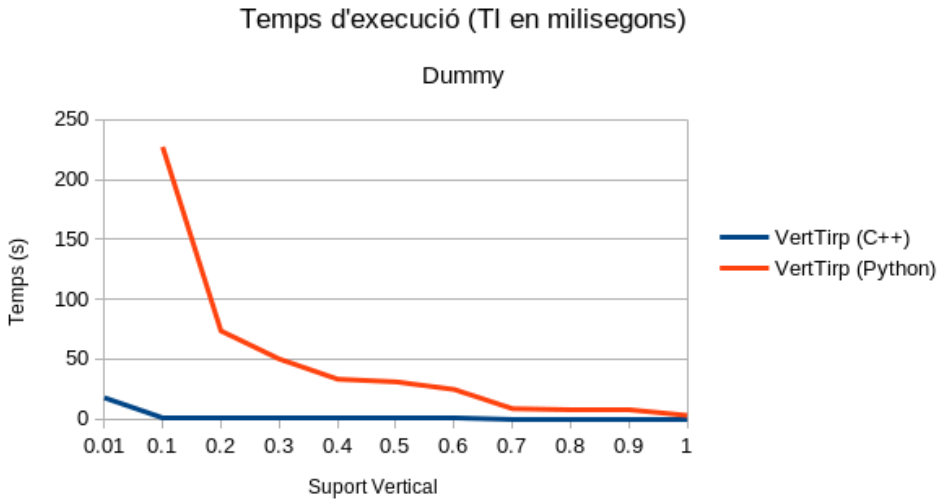


FIGURA 10.3: Temps d'execució del dataset *ASL* amb intervals de temps en milisegons i en *dummy*. (per veure les dades exactes consultar l'annex A.3)

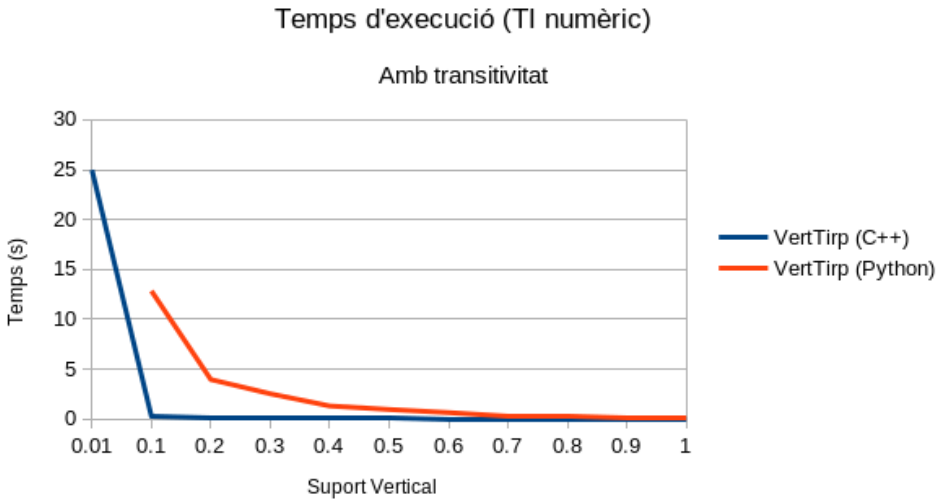


FIGURA 10.4: Temps d'execució del dataset *HAR* amb intervals de temps numèrics amb transitivitat. (per veure les dades exactes consultar l'annex A.4)

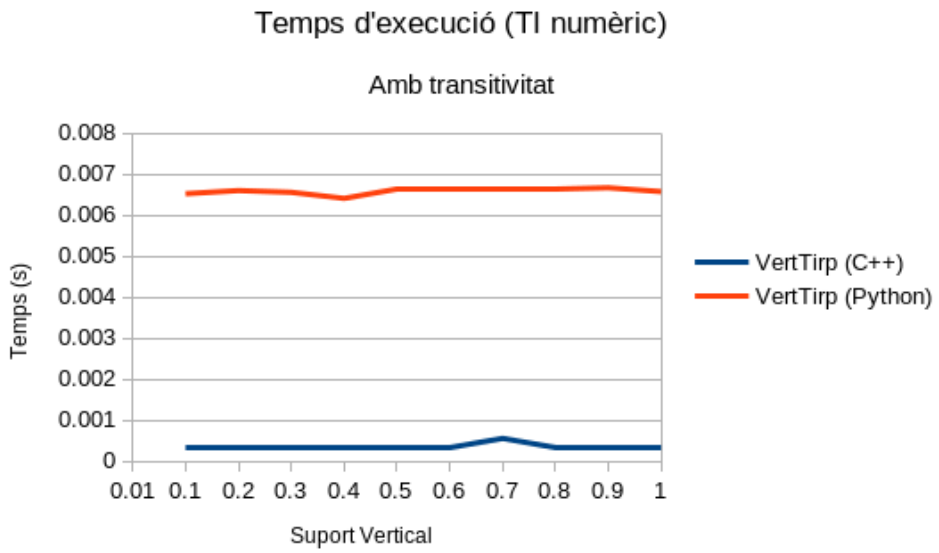


FIGURA 10.5: Temps d'execució del dataset *MAV* amb intervals de temps numèrics amb transitivitat. (per veure les dades exactes consultar l'annex A.5)

10.3 Temps d'execució en paral·lel

En les figures 10.6, 10.7 i 10.8 es mostren les proves que he realitzat comparant el rendiment amb multithreading utilitzant intervals de temps en segons. Ho he fet així per permetre una millor visualització de les diferències, ja que amb intervals de segons, el temps d'execució global és més elevat. No obstant això, aquesta secció no ha obtingut els resultats esperats.

En la figura 10.8, es pot observar una millora del temps de fins a un 90% amb l'addició de *threads*. Tot i això, aquest percentatge no representa una millora substancial, ja que en el millor dels casos només s'aconsegueix reduir la meitat del temps original. Cal tenir en compte que aquesta millora es basa en l'ús de 6 *threads* en comparació amb només un. A mesura que s'afegeixen més *threads*, la millora en el temps disminueix, i a partir de 6 *threads*, el temps fins i tot empitjora.

La meva CPU té 6 nuclis i 12 *threads*, de manera que sospito que el rendiment es deteriora a partir de 6 *threads* degut a que coincideixen dos *threads* en un mateix nucli, provocant així overhead (com s'explica a la secció 7.1). A més, no sempre es produeix una millora amb paral·lelització. Per exemple a la figura 10.6 la millora és molt poc substancial, més o menys al voltant d'un 5% (en algun cas no hi ha millora), i en el cas d'una execució en milisegons(10.9), és a dir, on ja de per si l'algoritme és força ràpid, l'addició de *threads* empitjora el temps total.

Per tant, encara que la paral·lelització pot ser beneficiosa en alguns casos, no és sempre la millor opció.

10.4 Ús de memòria

En la figura 10.10 he fet la mateixa prova que en la figura 10.1 analitzant la memòria màxima. En aquest apartat també hi ha una millora substancial, concretament l'algoritme en C++ requereix un 47% de la memòria que l'implementació amb python . A part de que de per si el C++ pugui ser més eficient amb la memòria, la raó principal és l'ús de punters per les instàncies de *Ti* i *Tirp* ja que com s'explica al capítol 8 evita fer moltes copies d'objectes.

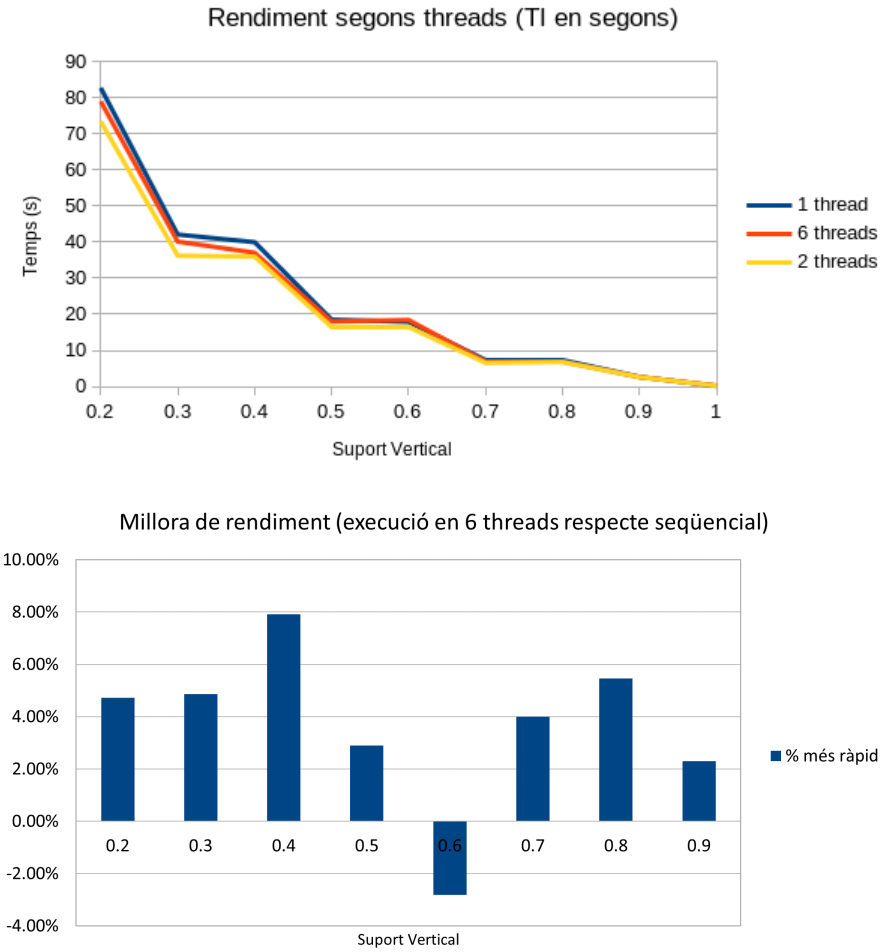


FIGURA 10.6: Diferència en el temps d'execució del vertTIRP en c++ amb el dataset *ASL* amb intervals de temps en segons, en funció del nombre de threads, i millora de temps entre execució en 6 threads respecte en seqüencial. (per veure les dades exactes consultar l'annex A.6)

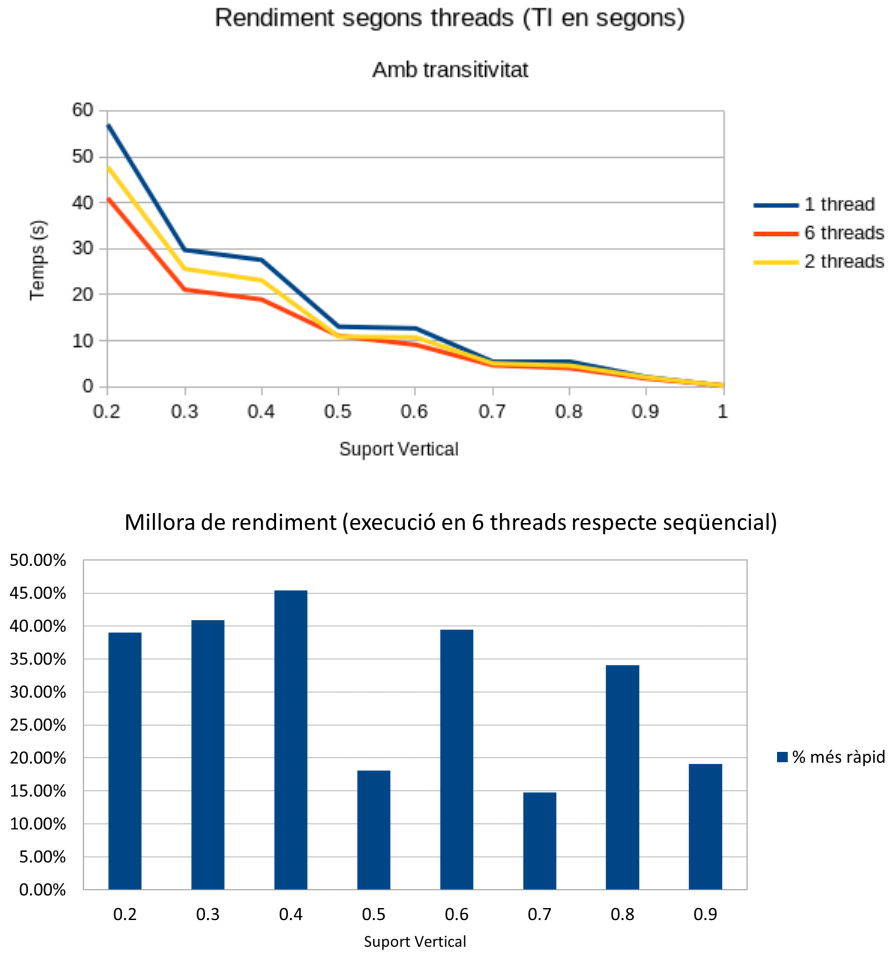


FIGURA 10.7: Diferència en el temps d'execució del vertTIRP en c++ amb el dataset *ASL* amb intervals de temps en segons i transitivitat, en funció del nombre de threads, i millora de temps entre execució en 6 threads respecte en seqüencial. (per veure les dades exactes consultar l'annex A.7)

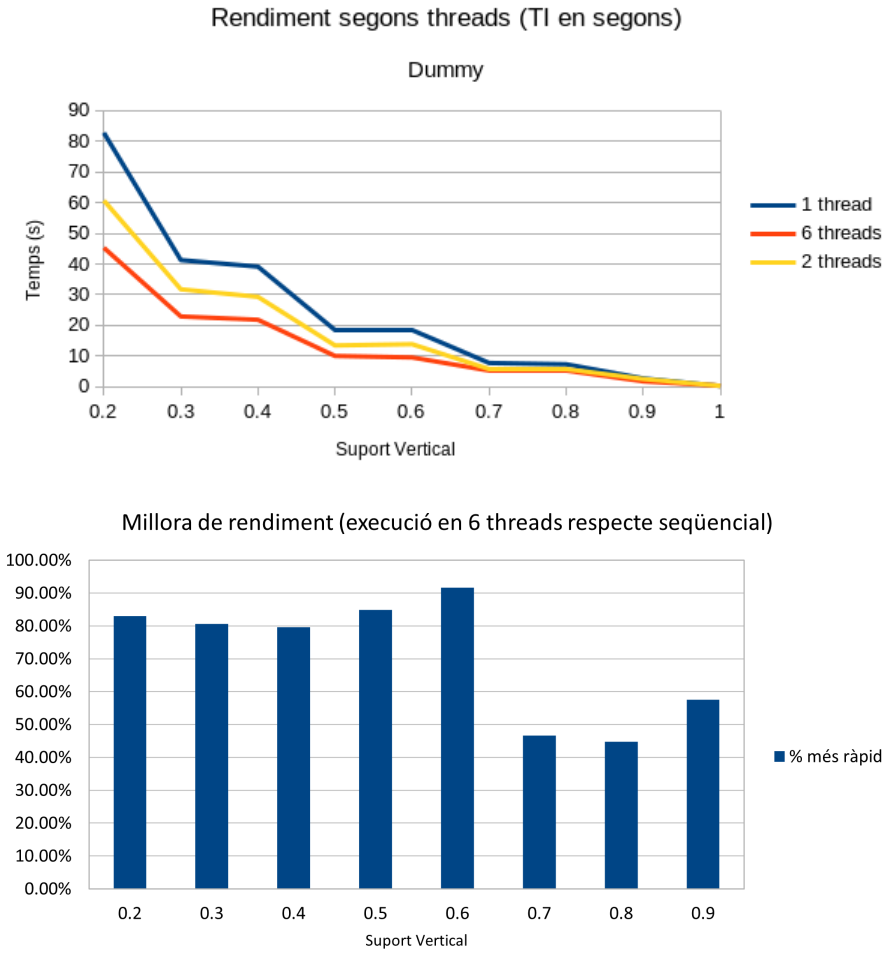


FIGURA 10.8: Diferència en el temps d'execució del vertTIRP en c++ amb el dataset *ASL* amb intervals de temps en segons i en *dummy*, en funció del nombre de threads, i millora de temps entre execució en 6 threads respecte en seqüencial. (per veure les dades exactes consultar l'annex A.6)

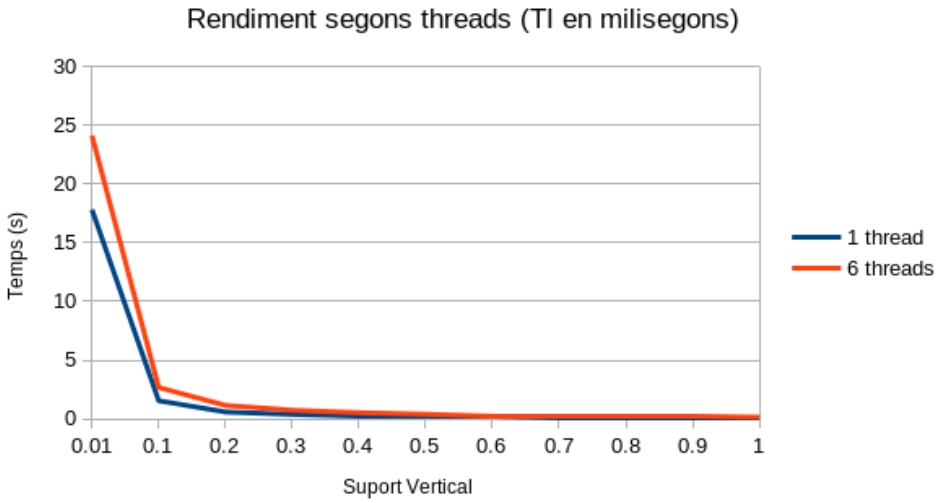


FIGURA 10.9: Diferència en el temps d'execució del vertTIRP en c++ amb el dataset *ASL* amb intervals de temps en milisegons, en 1 i 6 threads. (per veure les dades exactes consultar l'annex A.9)

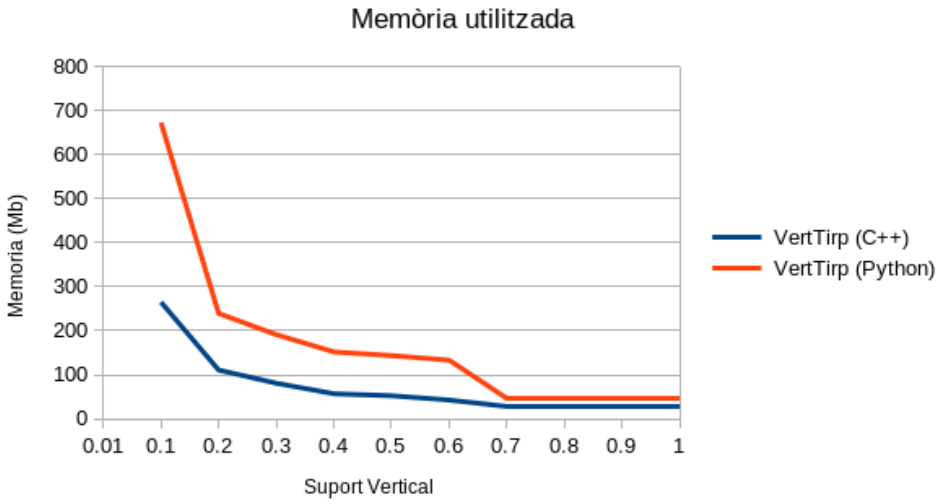


FIGURA 10.10: Diferència en l'ús de memòria amb el dataset *ASL* amb intervals de temps en milisegons. (per veure les dades exactes consultar l'annex A.10)

Capítol 11

Conclusions

En aquest document he detallat el procés d'adaptar un algoritme de *TIRP mining* a C++, l'objectiu era el de millorar el temps l'algoritme i el resultat ha estat positiu. Una de les principals complicacions, sobretot al principi va ser entendre bé el codi, abans de començar el projecte ja sabia que la persona que el va escriure ja no està a la universitat i per tant si no entenia alguna cosa no ho podria consultar. A més abans de començar a programar vaig haver d'estudiar-me el funcionament i conceptes de l'article [2]. No obstant el projecte ha continuat endavant i s'ha finalitzat.

He intentat que el codi sigui el màxim d'entenedor possible, seguint l'estructura que té l'original, els comentaris són els mateixos i els noms de les variables també. Tot i així, el C++ és un llenguatge més caòtic i alguna part pot ser complicada d'entendre. El codi està penjat en un repositori públic [1].

Els resultats d'adaptar l'algoritme són molt bons en quan a temps d'execució i ús de memòria, pot obtindre els mateixos resultats en un 1% del temps de l'original. Tot i així en l'apartat de paral·lelitzar, la millora no és gaire substancial i si hi ha un espai a millores seria en aquesta ultima part (veure capítol 12). En conclusió, crec que he assolit els objectius proposats i estic satisfet amb els resultats.

Capítol 12

Treball futur

En general els resultats del projecte han estat positius. No obstant, un apartat en el que hi ha espai a millora és la paral·lelització. Si bé com s'explica al capítol 10 sí que existeix una millora en paral·lelitzar l'algoritme, no és gaire important i, en alguns casos pot empitjorar.

Una possible orientació al problema podria ser analitzar en profunditat l'*scheduling* que utilitza la *CPU* i optimitzar el funcionament de l'algoritme per reduïr l'*overthread*. Una altra possiblilitat és que l'*overhead* sigui provocat per l'accés constant a memòria (ja que les estructures de dades es poden tornar molt grans) i una gestió de la *cache* i les comprovacions d'integritat d'aquesta podria ser interessant.

També es pot explorar l'implementació de la paral·lelització amb GPU. Jo aquesta opció la vaig descartar (vegeu secció 7.1) però potser existeix alguna manera de que sigui viable.

Apèndix A

Taules de resultats

Transitivity	FALSE
Dummy	FALSE
eps	0

vertical supp	VertTirp (C++)	VertTirp (Python)	% més ràpid
0.01	17.82		
0.1	1.54	166.71	10725.32%
0.2	0.58	53.3	9089.66%
0.3	0.38	36.07	9392.11%
0.4	0.23	22.13	9521.74%
0.5	0.21	22.13	10438.10%
0.6	0.15	18.85	12466.67%
0.7	0.11	6.03	5381.82%
0.8	0.1	6.42	6320.00%
0.9	0.1	6.17	6070.00%
1	0.08	2.26	2725.00%

Millora:	8213.04%

FIGURA A.1: Temps d'execució del dataset *ASL* amb intervals de temps en milisegons i la millora de temps del *vertTIRP* amb C++ respecte python.

Transitivity	TRUE
Dummy	FALSE
eps	0

vertical supp	VertTirp (C++)	VertTirp (Python)	% més ràpid
0.01	13.7		
0.1	1.35	106.46	7785.93%
0.2	0.56	39.61	6973.21%
0.3	0.37	27.68	7381.08%
0.4	0.21	17.83	8390.48%
0.5	0.28	15.91	5582.14%
0.6	0.13	12.17	9261.54%
0.7	0.08	5.08	6250.00%
0.8	0.09	5.03	5488.89%
0.9	0.1	5.05	4950.00%
1	0.06	2.38	3866.67%

Millora:	6592.99%

FIGURA A.2: Temps d'execució del dataset *ASL* amb intervals de temps en milisegons i transitivitat, i la millora de temps del *vertTIRP* amb C++ respecte python.

Transitivity	FALSE
Dummy	TRUE
eps	0

vertical supp	VertTirp (C++)	VertTirp (Python)	Millora
0.01	17.6		
0.1	1.39	227.35	16256.12%
0.2	0.53	73.44	13756.60%
0.3	0.37	49.98	13408.11%
0.4	0.25	33.16	13164.00%
0.5	0.2	30.89	15345.00%
0.6	0.16	24.56	15250.00%
0.7	0.1	8.48	8380.00%
0.8	0.09	8.12	8922.22%
0.9	0.09	8.23	9044.44%
1	0.06	2.83	4616.67%

Millora:	11814.32%

FIGURA A.3: Temps d'execució del dataset *ASL* amb intervals de temps en milisegons i en *dummy*, i la millora de temps del *vertTIRP* amb C++ respecte python.

Transitivity	TRUE
Dummy	FALSE
eps	0

vertical supp	VertTirp (C++)	VertTirp (Python)	Millora
0.01	24.9651012		
0.1	0.247639774	12.833297252655	0.01929666
0.2	0.076780022	3.93859004974365	6.3385883996
0.3	0.049068571	2.51052689552307	0.0305832302
0.4	0.023692192	1.28970980644226	0.0183701728
0.5	0.018368323	0.93816614151001	0.019578966
0.6	0.011955515	0.620847702026367	0.0192567597
0.7	0.006310839	0.288684606552124	0.0218606703
0.8	0.003802945	0.176034212112427	0.0216034426
0.9	0.002586539	0.121548891067505	0.0212798239
1	0.001010172	0.0297544002532959	0.0339503398

Millora:	0.654436846496

FIGURA A.4: Temps d'execució del dataset *HAR* amb intervals de temps numèrics amb transitivitat.

Transitivity	TRUE
Dummy	FALSE
eps	0

vertical supp	VertTirp (C++)	VertTirp (Python)	Millora
0.01			
0.1	0.000328894	0.0065305233001709	1885,60%
0.2	0.000329906	0.00660777091979981	1902,93%
0.3	0.000336968	0.00656628608703613	1848,64%
0.4	0.00033739	0.00641703605651856	1801,96%
0.5	0.000328383	0.00666117668151856	1928,48%
0.6	0.000329875	0.00665450096130371	1917,28%
0.7	0.000554154	0.00666499137878418	1102,73%
0.8	0.000331889	0.00663995742797852	1900,66%
0.9	0.000346847	0.00667572021484375	1824,69%
1	0.000341334	0.00658345222473145	1828,74%

Millora:	1794.17%

FIGURA A.5: Temps d'execució del dataset *MAV* amb intervals de temps numèrics amb transitivitat.

Transitivity	FALSE
Dummy	FALSE
eps	0

vertical supp	1 thread	6 threads	2 threads	% més ràpid
0.2	82.75	79.02	73.43	4.72%
0.3	42.09	40.14	36.25	4.86%
0.4	39.97	37.04	35.97	7.91%
0.5	18.52	18	16.42	2.89%
0.6	17.9	18.42	16.33	-2.82%
0.7	7.28	7	6.54	4.00%
0.8	7.34	6.96	6.81	5.46%
0.9	2.67	2.61	2.6	2.30%
1	0.13	0.26	0.18	-50.00%

Millora:	0.0366414740876

FIGURA A.6: Diferència en el temps d'execució del vertTIRP en c++ amb el dataset *ASL* amb intervals de temps en segons, en funció del nombre de threads.

Transitivity	TRUE
Dummy	FALSE
eps	0

vertical supp	1 thread	6 threads	2 threads	% més ràpid
0.2	57	41	47.77	39.02%
0.3	29.69	21.07	25.59	40.91%
0.4	27.51	18.92	23.09	45.40%
0.5	13.01	11.02	10.83	18.06%
0.6	12.65	9.07	10.7	39.47%
0.7	5.29	4.61	5.04	14.75%
0.8	5.35	3.99	4.55	34.09%
0.9	2.06	1.73	1.99	19.08%
1	0.13	0.23	0.18	-43.48%

Millora:	31.35%

FIGURA A.7: Diferència en el temps d'execució del vertTIRP en c++ amb el dataset *ASL* amb intervals de temps en segons i transitivitat, en funció del nombre de threads.

Transitivity	FALSE
Dummy	TRUE
eps	0

vertical supp	1 thread	6 threads	2 threads	% més ràpid
0.2	82.8	45.26	60.7	82.94%
0.3	41.24	22.84	31.7	80.56%
0.4	39.11	21.78	29.24	79.57%
0.5	18.41	9.96	13.4	84.84%
0.6	18.18	9.49	13.81	91.57%
0.7	7.67	5.23	5.6	46.65%
0.8	7.25	5.01	5.79	44.71%
0.9	2.63	1.67	2.45	57.49%
1	0.15	0.23	0.2	-34.78%

Millora:	71.04%

FIGURA A.8: Diferència en el temps d'execució del vertTIRP en c++ amb el dataset *ASL* amb intervals de temps en segons i en *dummy*, en funció del nombre de threads.

Transitivity	FALSE
Dummy	FALSE
eps	0

vertical supp	1 thread	6 threads
0.01	17.82	24.13
0.1	1.54	2.67
0.2	0.58	1.14
0.3	0.38	0.73
0.4	0.23	0.52
0.5	0.21	0.39
0.6	0.15	0.26
0.7	0.11	0.21
0.8	0.1	0.19
0.9	0.1	0.21
1	0.08	0.14

FIGURA A.9: Diferència en el temps d'execució del vertTIRP en c++ amb el dataset *ASL* amb intervals de temps en milisegons, en 1 i 6 threads.

Transitivity	FALSE
Dummy	FALSE
eps	0

vertical supp	VertTirp (C++)	VertTirp (Python)	Millora
0.01			
0.1	265.01064	672.823	39.39%
0.2	110.8	239.326	46.30%
0.3	80.76	191.145	42.25%
0.4	56.84	151.849	37.43%
0.5	52.36	143.602	36.46%
0.6	42.62	133.059	32.03%
0.7	28.43	46.952268	60.55%
0.8	28.43	46.919	60.59%
0.9	28.43	46.919	60.59%
1	28.43	46.919	60.59%

Millora:	47.62%

FIGURA A.10: Diferència en l'ús de memòria (*Mb*) amb el dataset *ASL* i intervals de temps en milisegons.

Apèndix B

Datasets

	A	B	C	D	E	F	G	H	I	J
1	Consultant	Main_New_Gloss	D_Start_HS	N-D_Start_HS	D_End_HS	N-D_End_HS	Passive_Arm	Start	End	Session_Scene
2	Liz	TWENTY	crvd-L		flat-G		N	2635	2661	ASL_2008_01_112
3	Tyler	TWENTY	L		flat-G		N	2400	2480	ASL_2008_05_12a1
4	Naomi	TWENTY	L		baby-O		N	2279	2353	ASL_2008_08_041
5	Brady	TWENTY	L		flat-G		N	5170	5198	ASL_2011_06_08_Brady5
6	Liz	ALONE	1		1		N	3707	3782	ASL_2008_02_0135
7	Liz	ALONE	1		1		N	2890	2936	ASL_2008_02_2932
8	Tyler	ALONE	1		1		N	3801	3830	ASL_2008_05_29b9
9	Naomi	ALONE	1		1		N	4065	4141	ASL_2008_08_1329
10	Brady	ALONE	1		1		N	1430	1498	ASL_2011_06_14_Brady42
11	Brady	ALONE	1		1		N	3186	3211	ASL_2011_07_19_Brady83
12	Lana	ALONE	1		1		N	2076	2120	ASL_2006_10_102
13	Dana	ALONE	1		1		N	2431	2476	ASL_2007_05_245

FIGURA B.1: Format del dataset *asl*[30] abans de ser adaptat per al *vertTIRP*.

	A	B	C	D	E	F
1	BodyAcc	GravityAcc	BodyGyro	subject	Activity	time
2	b	a	b	1	STANDING	2010-01-01 10:00:00.000
3	b	a	b	1	STANDING	2010-01-01 10:00:02.560
4	b	a	b	1	STANDING	2010-01-01 10:00:05.120
5	b	a	b	1	STANDING	2010-01-01 10:00:07.680
6	b	a	b	1	STANDING	2010-01-01 10:00:10.240
7	b	a	b	1	STANDING	2010-01-01 10:00:12.800
8	b	a	b	1	STANDING	2010-01-01 10:00:15.360
9	b	a	b	1	STANDING	2010-01-01 10:00:17.920
10	b	a	b	1	STANDING	2010-01-01 10:00:20.480
11	b	a	b	1	STANDING	2010-01-01 10:00:23.040

FIGURA B.2: Format del dataset *har*[29] abans de ser adaptat per al *vertTIRP*.

```
data hora ampm lloc1 lloc2 sensor estat
02/04/2003 10:36:0 PM (Conference Room) J11 ON
02/04/2003 10:36:0 PM (Conference Room) J11 ON
02/04/2003 10:36:0 PM (Conference Room) J11 ON
02/04/2003 10:36:0 PM (Conference Room) J11 ON
02/04/2003 10:36:0 PM (Living Room) H13 OFF
02/04/2003 10:36:0 PM (Living Room) H13 OFF
```

FIGURA B.3: Format del dataset *mav*[28] abans de ser adaptat per al *vertTIRP*.

LISTING B.1: Comanda SQL que adapta el dataset *asl*[30] (annex B.1) per ser interpretat per l'algoritme

```
SELECT
session_scene AS sid,
Start AS start_time,
End AS end_time,
CONCAT(Main_New_Gloss, D_Start_HS, D_End_HS, Passive_Arm) AS value
FROM datasets.asl
order BY session_scene,Main_New_Gloss, D_Start_HS, D_End_HS, Passive_Arm
```

LISTING B.2: Comanda SQL que adapta el dataset *har*[29] (annex B.2) per ser interpretat per l'algoritme

```
with A AS (
    SELECT Subject,BodyAcc,GravityAcc,BodyGyro,Activity,TIME,
     LAG(Activity, 1) OVER (ORDER BY Subject,BodyAcc,GravityAcc,BodyGyro,
    time) AS Previ
  FROM datasets.har
), B AS (
    SELECT Subject,BodyAcc,GravityAcc,BodyGyro,Activity,time,
     SUM(CASE WHEN Activity = Previ THEN 0 ELSE 1 END)
                OVER (ORDER BY Subject,BodyAcc,GravityAcc,BodyGyro,time)
    AS Ranker
  FROM A
)
SELECT SUBJECT AS sid,UNIX_TIMESTAMP(min(TIME)) AS start_time,
COALESCE(
LEAD(UNIX_TIMESTAMP(MIN(TIME))) OVER( partition by Subject,BodyAcc,
    GravityAcc,BodyGyro ORDER BY Subject,BodyAcc,GravityAcc,BodyGyro,TIME),
UNIX_TIMESTAMP(max(TIME))) AS end_time,
CONCAT(BodyAcc,GravityAcc,BodyGyro,Activity) AS value
FROM B
GROUP BY Subject,BodyAcc,GravityAcc,BodyGyro,Activity,Ranker
ORDER BY Subject,BodyAcc,GravityAcc,BodyGyro,time
```

LISTING B.3: Comanda SQL que adapta el dataset *mav*[28] (annex B.3) per ser interpretat per l'algoritme

```
with A AS (
    SELECT *,
     LAG(estat) OVER (partition by lloc,sensor ORDER BY lloc,sensor,DATA
    asc ) AS Previ
  FROM datasets.mav
  GROUP BY lloc,sensor,DATA
), B AS (
    SELECT *,
     SUM(CASE WHEN estat = Previ THEN 0 ELSE 1 END)
                OVER (ORDER BY lloc,sensor,data rows between unbounded
    preceding and current row) AS Ranker
  FROM A
)
SELECT  lloc AS sid,
UNIX_TIMESTAMP(min(data)) AS start_time,
COALESCE(
LEAD(UNIX_TIMESTAMP(MIN(data))) OVER( partition by lloc,sensor ORDER BY
    lloc,sensor,data),
UNIX_TIMESTAMP(max(DATA))
) AS end_time,
CONCAT(lloc,sensor,estat) AS value
from B
GROUP BY lloc,sensor,Ranker
ORDER BY lloc,sensor,DATA
```

Apèndix C

Manual d'usuari i/o instal·lació

Per executar l'algoritme, jo he utilitzat el compilador *g++* [13] en un linux, però també funciona en windows. Per compilar només s'han s'han d'escriure les comandes següents:

```
$g++ -c main.cpp VertTIRP.cpp Allen.cpp TI.cpp Utils.cpp Global.h
    AllenRelationsEPS.cpp PairingStrategy.cpp Relation.cpp TIRP.cpp
    TIRPstatistics.cpp VertTirpSidList.cpp VertTirpNode.cpp Chrono.cpp
```

```
$g++ -o vert_tirp_c -O3 -fopenmp main.o VertTIRP.o Allen.o TI.o Utils.o
    AllenRelationsEPS.o PairingStrategy.o Relation.o TIRP.o TIRPstatistics.
    o VertTirpSidList.o VertTirpNode.o Chrono.o
```

Amb això generarem l'executable *vert_tirp_c*. Els valors d'entrada com el nom del fitxer, valors mínims de suport, mode de temps, etc. Es defineixen en el fitxer *main.cpp* com a variables globals. En cas de que no vulguem utilitzar la API *openMp* només cal treure el flag *-fopenmp* de la segona comanda.

LISTING C.1: Exemple d'execució

```
$.\vert_tirp_c 12
```

Per especificar el nombre de threads que volem que s'utilitzin es pot indicar com a paràmetre tal com es veu en l'exemple C.1 i si no se li passa res, utilitza la meitat del nombre màxim de *threads* per defecte. Al executar l'algoritme es generarà un fitxer *.cvc* com el de la figura C.1 amb el resultat.

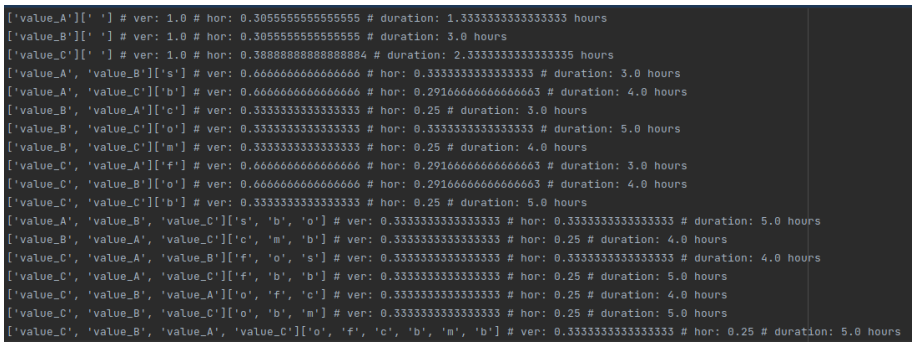


FIGURA C.1: Fitxer de sortida a partir de l'exemple 8.3

Bibliografia

[1] Pau Calvera. *VertTIRP C++*. Available at https://github.com/Pcalvera/vertTIRP_c, consultat: 11 de maig de 2023. 2022.

[2] Natalia Mordvanyuk, Beatriz López i Albert Bifet. "vertTIRP: Robust and efficient vertical frequent time interval-related pattern mining". A: *Expert Systems with Applications* 168 (2021). Consultat: 11 de maig de 2023, pàg. 114276.

[3] *CLion*. https://www.jetbrains.com/es-es/clion/. Consultat: 11 de maig de 2023.

[4] *PyCharm*. https://www.jetbrains.com/pycharm/. Consultat: 11 de maig de 2023.

[5] *Bitbucket*. https://bitbucket.org/. Consultat: 11 de maig de 2023.

[6] *GitHub*. https://github.com/. Consultat: 11 de maig de 2023.

[7] *Overleaf*. https://www.overleaf.com/. Consultat: 11 de maig de 2023.

[8] *LibreOffice Calc*. https://es.libreoffice.org/descubre/calc/. Consultat: 11 de maig de 2023.

[9] *Diagrams.net*. https://app.diagrams.net/. Consultat: 11 de maig de 2023.

[10] *Valgrind*. https://valgrind.org/. Consultat: 11 de maig de 2023.

[11] *GanttProject*. https://www.ganttproject.biz/. Consultat: 11 de maig de 2023.

[12] *Diagrama de Gantt*. https://ca.wikipedia.org/wiki/Diagrama_de_Gantt. Consultat: 11 de maig de 2023.

[13] *GCC, the GNU Compiler Collection*. https://gcc.gnu.org/. Consultat: 11 de maig de 2023.

[14] *Python*. https://www.python.org/. Accessed: May 11, 2023.

[15] *OpenMp*. https://www.openmp.org/. Consultat: 11 de maig de 2023.

[16] *MySQL*. https://www.mysql.com/. Consultat: 11 de maig de 2023.

[17] *HeidiSQL*. https://www.heidisql.com/. Consultat: 11 de maig de 2023.

[18] *Metodologia àgil*. https://ca.wikipedia.org/wiki/Metodologia_Ãăgil. Consultat: 11 de maig de 2023.

[19] *Scrum*. https://ca.wikipedia.org/wiki/Scrum. Consultat: 11 de maig de 2023.

[20] *Què és Kanban*. https://asana.com/es/resources/what-is-kanban. Consultat: 11 de maig de 2023.

[21] *Metodologia Lean en l'empresa*. https://www.ekon.es/blog/metodologia-lean-empresa/. Consultat: 11 de maig de 2023.

[22] *Programació extrema*. https://ca.wikipedia.org/wiki/ProgramaciÃş_extrema. Consultat: 11 de maig de 2023.

[23] *NVIDIA CUDA Toolkit*. https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html. Consultat: 11 de maig de 2023.

[24] *The khronos group: Opencl*. https://www.khronos.org/opencl/. Consultat: 11 de maig de 2023.

[25] *Chat GPT*. https://cat-gpt.com/. Consultat: 11 de maig de 2023.
[26] *GitHub Copilot*. https://github.com/features/copilot. Consultat: 11 de maig de 2023.
[27] *CodeConvert AI*. https://www.codeconvert.ai/. Consultat: 11 de maig de 2023.
[28] Full Circle Security. *Smart Home Datasets and a Realtime Internet Connected Home*. https://fullcirclesecurity.org/2019/06/03/smart-home-datasets-and-a-realtime-internet-connected-home/. Consultat: 11 de maig de 2023.
[29] UCI Machine Learning Repository. *Human Activity Recognition Using Smartphones*. https://archive.ics.uci.edu/ml/datasets/human%2Bactivity%2Brecognition%2Busing%2Bsmartphones. Consultat: 11 de maig de 2023.
[30] Universitat de Boston. *ASLLRP Database*. https://www.bu.edu/asllrp/. Consultat: 11 de maig de 2023.
[31] *C++ Standard Library*. https://en.cppreference.com/w/cpp/header. Consultat: 11 de maig de 2023.
[32] *Standard library header <memory>*. https://en.cppreference.com/w/cpp/header/memory. Consultat: 11 de maig de 2023.
[33] James F Allen. "Maintaining knowledge about temporal intervals". A: *Communications of the ACM* 26.11 (1983). Consultat: 11 de maig de 2023, pàg. 832 - 843.