Universitat de Girona
Escola Politècnica Superior

Grau en Enginyeria Informàtica

PROJECTE FINAL DE GRAU

Mapes de Cobertura en Entorns amb Obstacles

Autor:
Oriol Balló Gimbernat

Tutors:
Dr. Narcís Coll
Dra. Marta Fort

MEMÒRIA

Convocatòria:
Juny 2023

Departament :
Informàtica, Matemàtica Aplicada i Estadística

Projecte: Projecte Final de Grau
Document: Memòria
Títol: Mapes de Cobertura en Entorns amb Obstacles
Autor: Oriol Balló Gimbernat
Data: Juny 2023

Estudi:
Grau en Enginyeria Informàtica
Universitat de Girona

Supervisor 1:
Dr. Narcís Coll
Universitat de Girona
Email: narcis.coll@udg.edu

Supervisor 2:
Dra. Marta Fort
Universitat de Girona
Email: marta.fort@udg.edu

Índex

Capítol 1

Introducció

Aquest projecte és una continuació de la recerca realitzada al GILab de la Universitat de Girona pel doctor Narcís Coll i la doctora Marta Fort sobre problemes de cobertura de la branca de la informàtica i matemàtiques coneguda com a geometria computacional.

Els problemes de cobertura tracten de, donada una distribució de serveis, determinar quines regions d'un domini queden cobertes. Hi ha variants d'optimització d'aquest problema on s'afegeixen graus de llibertat, per exemple: intentant trobar la millor distribució d'un nombre fix de serveis o la millor amb el mínim nombre possible d'aquests. En tots els casos la solució és un mapa de cobertura, una representació gràfica que ens permet determinar quines són les regions cobertes per com a mínim un servei.

En aquest tipus de problemes es considera que un punt està cobert per un servei si existeix algun camí de longitud menor o igual al radi d'abast d'aquest que els connecti. En entorns sense obstacles això implica que la cobertura de cada servei sigui una circumferència del mateix radi que aquest, ja que tots els camins que arribin al servei seran línies rectes de la mateixa longitud. Quan treballem en entorns amb obstacles ja no podem assegurar que l'àrea coberta per qualsevol servei sigui sempre una circumferència. En aquest cas podem trobar-nos camins que hagin de vorejar obstacles per tal d'arribar al seu objectiu i, per tant, passin de ser rectes a poligonals que creuen per les cantonades dels obstacles.

Els mapes de cobertura tenen un ampli rang d'aplicacions tant al sector públic com al privat, ja que es poden aplicar a qualsevol classe de servei que tingui un cert rang d'abast, tant real o com imposat, des de punts de càrrega de robots on podem voler que cada robot operi com a molt X distància d'un d'aquests, p. ex. robots Roomba, fins a *kits* de primers auxilis que volem que estiguin a un temps/distància raonable de cada sala d'un edifici. En ambdós casos estem buscant un mapa de cobertura que maximitzi aquesta. Addicionalment, aquest ens permetrà prendre decisions sobre el nombre de serveis que necessitem i la qualitat d'aquests (rang d'abast), ja que a partir del mapa podem determinar que en necessitem més o que amb menys ja en faríem prou perquè hi ha un alt nivell de coincidència.

En aquest projecte es presenta una solució per l'obtenció de mapes de cobertura en entorns en obstacles basada en una modificació de l'algoritme de Bellman-Ford en paral·lel que aprofita la capacitat de còmput de la GPU i una tècnica d'optimització basada en un algoritme genètic per trobar distribucions de serveis que maximitzin la cobertura.

1.1 Propòsit i objectius

El primer i principal propòsit del projecte és desenvolupar un programa que utilitzant les capacitats computacionals de la GPU permeti, eficientment: calcular mapes de cobertura d'entorns amb obstacles i trobar distribucions de serveis que en maximitzin la cobertura. Alhora, el segon propòsit és que la presentació de resultats sigui clara i fàcilment interpretable.

Els objectius específics són:

- dissenyaer i implementar, utilitzant la GPU, un algoritme que permeti calcular mapes de cobertura en entorns en obstacles.
- implementar mitjançant algoritmes genètics l'obtenció de solucions òptimes que maximitzin la cobertura per un cert nombre de serveis.
- desenvolupar una interfície gràfica que permeti fer servir diferents paràmetres d'execució i visualitzar els resultats.

1.2 Organització de la memòria

Respecte a l'estructura de la memòria, en tractar-se d'un projecte de recerca, s'ha considerat necessari destinar un capítol sencer a l'anàlisi i disseny de la solució desenvolupada. Per això s'ha decidit dividir el capítol vuit en dos. En el capítol 8 es presenta l'anàlisi i disseny de la solució i s'ha afegit el capítol 9 en el que s'explica l'anàlisi i disseny del sistema.

Addicionalment, degut a les característiques del sistema desenvolupat el capítol 11 només s'exposen els resultats. El contingut corresponent a la implantació queda relegat al manual d'usuari, ja que només consisteix a baixar i executar un executable de Windows.

Capítol 2

Estudi de viabilitat

Per fer possible el desenvolupament podem dividir els requisits en tecnològics, on ens cal un ordinador amb targeta gràfica NVIDIA compatible amb CUDA, i en requisits de viabilitat, on hem d'avaluar si el problema a tractar és adequat per ser resolt amb una GPU.

Des del punt de vista tecnològic jo personalment ja disposava d'un ordinador amb una targeta gràfica compatible amb CUDA i, per tant, no hi ha hagut cap cost associat al hardware necessari. Un dels principals avantatges de treballar amb CUDA és que no cal disposar d'una targeta moderna per començar a desenvolupar, en el meu cas he pogut fer tot el projecte sense cap problema amb una targeta de l'any 2017. Alternativament, en cas de voler un millor rendiment i les últimes funcionalitats llavors sí que hauríem d'adquirir una targeta més moderna on els costos ja es poden elevar fins als milers d'euros. El cost d'entrada per programar en CUDA es pot pràcticament adaptar pràcticament a qualsevol pressupost i varia segons els requisits que tinguem. Finalment, totes les llibreries utilitzades (Capítol 7) són lliures, de codi obert i no tenen cap cost associat, per tant, no és necessària cap inversió econòmica pel desenvolupament del projecte.

Des del punt de vista d'investigació caldria avaluar si el problema que estem tractant és paral·lelitzable, ja que en cas contrari utilitzar la GPU no ens aportaria cap benefici. Tal com es podrà veure en detall a la implementació de l'algoritme (Capítol 10) i en el seu disseny (Capítol 8) cada punt de l'entorn depèn només dels seus 8 veïns (Figura 2.1) més propers i, per tant, podem tractar la majoria de punts del domini de manera independent. Aleshores té sentit utilitzar la GPU, ja que el problema és altament paral·lelitzable.

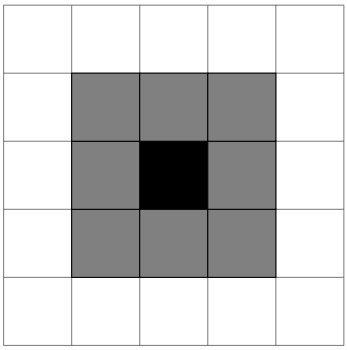


FIGURA 2.1: Punt de l'entorn (negre), veïns del punt (gris) i altres (blanc)

Respecte a la viabilitat econòmica, tenint en compte que el present projecte és de recerca i podria formar una petita part d'una tesi doctoral tenim el següent: el sou mínim per estudiants de doctorat a Espanya està establert a 16.422 € bruts anuals, si dividim aquest entre el nombre d'hores laborables que hi ha en un any obtenim $\sim 9,04$ €/h. Considerant que el projecte ha durat quatre mesos i de mitjana he destinat unes cinc hores al dia el cost aproximat seria de 3.600 €.

Feina	**Cost per hora (€)**	**Temps destinat (h)**	**Cost total (€)**
Projecte de recerca	9,04	400	3.617,18

TAULA 2.1: Pressupost estimat

En el cas que fos necessari adquirir una targeta gràfica NVIDIA, o ordinador, compatible amb CUDA llavors s'hauria d'afegir al cost i el pressupost augmentaria entre $150 - 500$€.

Capítol 3

Metodologia

Com tot projecte de recerca una de les coses més importants a tenir en compte en desenvolupar el projecte és la incertesa sobre el que pot funcionar i el que no, ja que en un gran nombre de casos no podem afirmar que una idea és correcte fins que no la tenim implementada dins de l'estructura global del programa. Addicionalment en aquest projecte calia tenir en compte que havia d'aprendre CUDA des de zero i, per tant, requeria un cert nivell de flexibilitat a l'hora de planificar i organitzar.

Donades les característiques mencionades vaig escollir utilitzar Scrum, que tot i estar pensada per treballar en equips de 7 ± 2 integrants, és fàcilment adaptable a altres situacions. Aquesta metodologia consisteix en un procés iteratiu de: obtenció de *feedback*, incorporació de *feedback* mitjançant creació de tasques concretes, planificació i execució de *sprints* on es treballa en un subconjunt de tasques.

En el cas particular d'aquest projecte l'obtenció de *feedback* es feia mitjançant reunions amb els tutors que segons la complicació del *sprint* present variaven de bisetmanals a quinzenals. D'aquestes en sortien noves tasques i se n'escollia un subconjunt per implementar abans de la següent reunió, que es produïa un cop totes estaven implementades o apareixia una complicació major. Així, cada *sprint* tenia una durada flexible adequada a les necessitats presents del projecte.

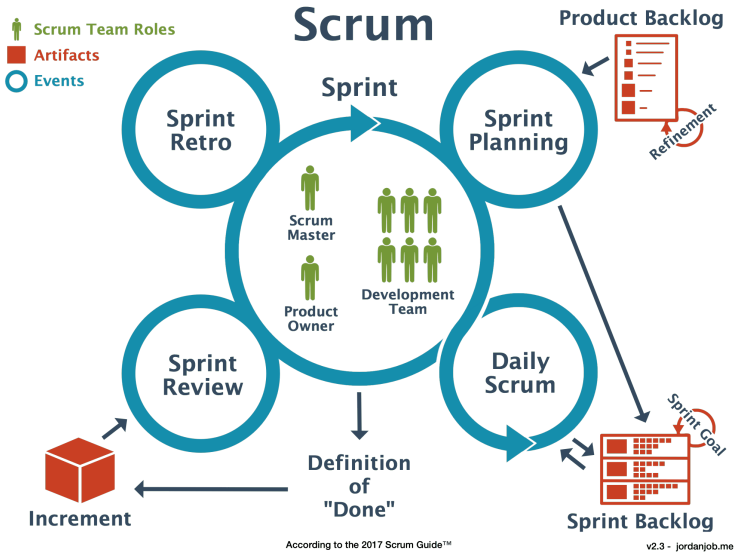


FIGURA 3.1: Digrama de la metodologia Scrum

Capítol 4

Planificació

Tenint en consideració el punt de partida del projecte i la necessitat d'aprendre CUDA des de zero, la planificació es va estructurar en tres etapes principals:

- Aprenentatge de programació de GPU amb CUDA.
- Disseny de solucions, desenvolupament del sistema i implementació dels diferents algoritmes.
- Elaboració de la memòria.

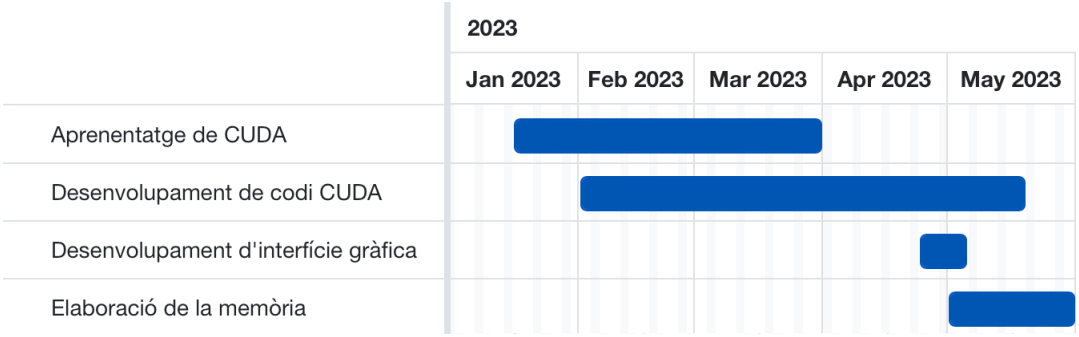


FIGURA 4.1: Planificació del projecte

Encara que l'aprenentatge de CUDA s'ha mantingut constant al llarg de tot el projecte, al diagrama només es mostra el temps explícitament destinat a aquest.

Durant les fases de disseny, desenvolupament i redacció, es va seguir la metodologia especificada a l'apartat anterior. El fet que aquest projecte fos de recerca, on era necessari dissenyar algorismes per resoldre el problema plantejat, va contribuir al fet que la fase de disseny de solucions fos llarga i extensa.

Capítol 5

Marc de treball i conceptes previs

5.1 Marc de treball

Tal com s'ha mencionat a la introducció (Capítol 1) aquest projecte està situat en el marc de la geometria computacional que forma part de la recerca realitzada al GiLab del departament IMAE de la Universitat de Girona.

Aquest projecte, partint des de zero, pretén implementar una variació de l'algoritme de Bellman-Ford amb la GPU que generalitza mètodes existents de càlcul de mapes de cobertura per entorns amb obstacles i utilitzar aquest per trobar, mitjançant un algoritme genètic, mapes que maximitzin la cobertura donat un conjunt de serveis.

La Doctora Marta Fort i el Doctor Narcís Coll són investigadors del departament d'IMAE de la UdG i han col·laborat proporcionant l'idea general de l'algoritme de càlcul de mapes de cobertura i guiant la seva implementació, correcció i actualització al llarg del projecte.

A continuació s'expliquen tots els conceptes necessaris per entendre la part tècnica del projecte.

5.2 Programació de GPU

Les primeres GPU tal com les coneixem avui en dia van ser primer desenvolupades a mitjans dels anys noranta per empreses com 3Dfx, ATI i NVIDIA. L'objectiu d'aquestes era satisfer la demanda que hi havia per una major capacitat de processat gràfic procedent de diferents indústries. Aquests nous tipus de processadors gràfics estaven específicament dissenyats per complementar la CPU encarregant-se dels càlculs en paral·lel, que tant perjudicaven aquesta, en tasques de processat 3D.

La programació de GPU, coneguda com a GPGPU, no es va començar a popularitzar fins a principis dels dos mil quan es van desenvolupar les primeres versions dels *frameworks* CUDA (*Compute Unified Device Architecture*) i OpenCL (*Open Computing Language*). Aquests van fer accessible la GPGPU a major públic mitjançant l'estandardització i abstracció dels detalls de baix nivell.

L'augment de la capacitat de còmput de les GPU conjuntament amb la nova accessibilitat a GPGPU va provocar que no es tardés a trobar aplicacions fora de l'àmbit de processament de gràfics. Avui en dia s'utilitzen pràcticament en qualsevol àmbit

de computació d'alt rendiment, des de la intel·ligència artificial i simulacions científiques fins a videojocs i realitat augmentada.

5.3 Programació i Arquitectura CUDA

CUDA és el *framework* de programació de targetes gràfiques exclusiu de NVIDIA que permet utilitzar la capacitat computacional de les seves targetes gràfiques per accelerar càlculs i aplicacions. La seva sintaxi és molt semblant a `C/C++` i alhora ens proporciona una API que simplifica tota la gestió de baix nivell necessària per comunicar-se treballar amb la GPU.

L'arquitectura CUDA (Figura 5.1) està formada per unitats d'execució (Figura 5.2) anomenades *Streaming Multiprocessor*, abreviat SM, cada un d'aquests disposa d'una sèrie de CUDA cores, també coneguts com a *Streaming Processors*, abreviat SP. Així la capacitat de còmput d'una GPU de NVIDIA ve donada pel nombre de *Streaming Multiprocessor* i les seves corresponents característiques que a la pràctica ens indiquen amb quants *threads* podem treballar, quanta memòria de cada tipus tenim disponible...

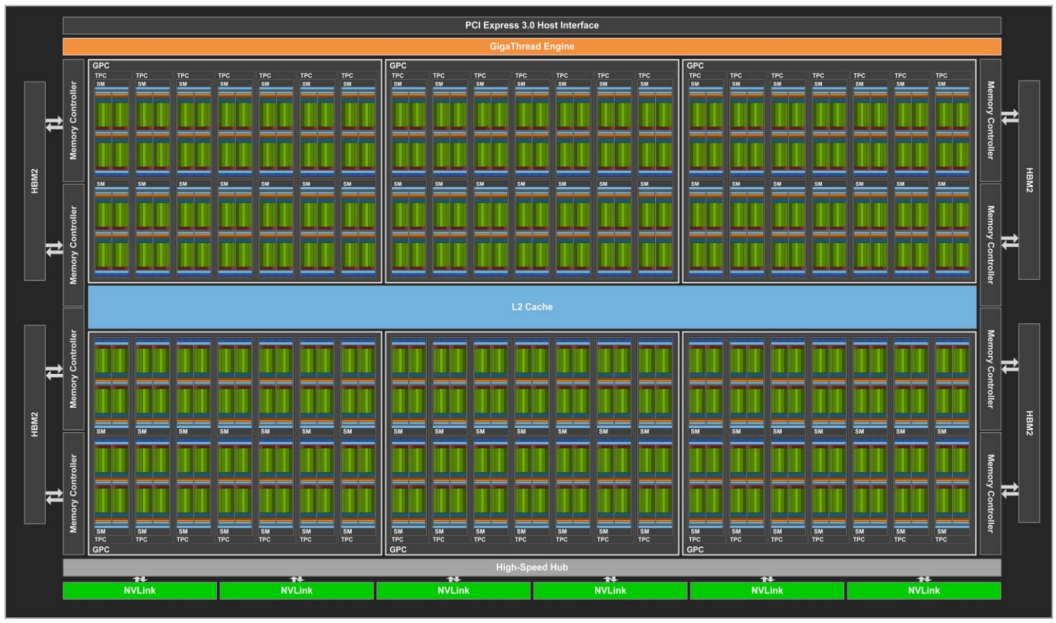


FIGURA 5.1: Arquitectura d'una GPU de NVIDIA de la generació Volta

Quan treballem amb CUDA no podem directament enviar funcions arbitràries a la GPU per ser executades directament sinó que hem de crear un tipus especial de funcions anomenades *kernels*. Aquestes són força semblants a les típiques funcions de C/C++, però tenen accés a un conjunt de variables exclusives de CUDA (vegeu Ansorge 2022 per a més detalls). La principal diferència entre funcions i *kernels* és que en cridar un *kernel* hem d'especificar dos valors, *threads* per bloc i blocs per *grid*, que dictaran com es paral·lelitza el codi. Aquests *threads* i blocs (conjunts de *threads*) formen part d'una jerarquia (Figura 5.3) que dictamina, entre d'altres, quins *threads* es poden comunicar.

FIGURA 5.2: Arquitectura processador SM d'una GPU de NVIDIA de la generació Volta

Els *threads* per bloc ens indiquen quants *threads* tindrà cada bloc i les seves dimensions, per exemple `dim3 threadsPerblock(16,16)` indica blocs de dues dimensions de 256 *threads* i `dim3 threadsPerblock(256)` blocs d'una dimensió de 256 *threads*. Finalment, els blocs per grid indiquen quants blocs dels anteriors posar. Normalment, es busca emplenar el grid i, per tant, es calcula a partir dels *threads* per bloc i les dimensions de l'*input*, per exemple com aquí:

```
dim3 blocsPerGrid((cols + threadsPerblock.x - 1) / threadsPerblock.x, (rows
+ threadsPerblock.y - 1) / threadsPerblock.y)
```

Aquests dos valors s'han d'adaptar al problema tractat, ja que poden afectar l'eficiència i rendiment. En aquest projecte s'utilitzen blocs de dues dimensions perquè s'adapta a l'*input* que tenim (imatges) però per altres problemes pot ser millor treballar amb blocs de diferents dimensions, per exemple en simular fluids blocs de tres dimensions són més adequats.

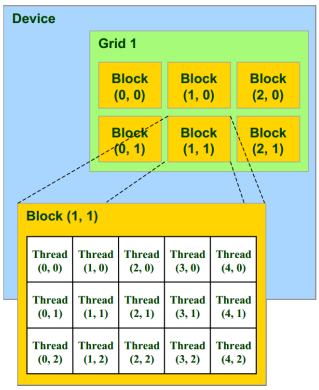


FIGURA 5.3: Jerarquia de *threads*, *blocs* i *grid*

5.3.1 Tipus de Memòria

Les GPU compatibles amb CUDA disposen de diferents tipus de memòria (Figura 5.4) que varien en mida, velocitat, permisos d'escriptura i lectura, sincronització... vegeu "NVIDIA Official CUDA C++ Programming Guide" 2023 i Sanders i Kandrot 2011 per més detalls.

La memòria global és la memòria de major capacitat que hi ha a la GPU, permet tant escriptura com lectura i és accessible per *threads*, blocs, CPU i GPU. És més lenta que la resta de memòries, però és adequada per dades d'entrada i sortida que s'han d'utilitzar en múltiples blocs i/o crides de *kernel*.

La memòria de textures és la memòria tradicionalment destinada per emmagatzemar textures gràfiques. Com a conseqüència és de només lectura, té una alta amplada de banda i un mecanisme de memòria cau que explota la localitat espacial que apareix tradicionalment a les textures. Normalment, s'empra quan tenim dades de lectura que presenten aquesta localitat com per exemple imatges i senyals.

La memòria constant és la memòria destinada a emmagatzemar dades de caràcter constant. Disposa d'una elevada amplada de banda conjuntament amb una baixa latència gràcies a una memòria cau que permet fer accessos molt ràpids.

La memòria *shared* és la memòria que comparteixen els *threads* d'un mateix bloc. És tant de lectura com d'escriptura i permet la col·laboració i sincronització de *threads*. Disposa d'un espai limitat però d'una major amplada de banda i menor latència que la memòria global.

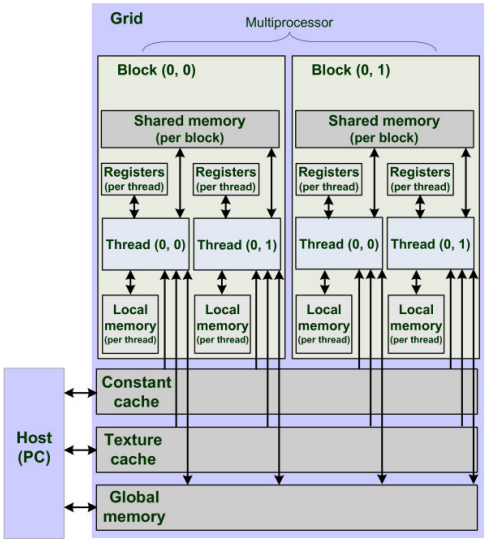


FIGURA 5.4: Jerarquia i tipus de memòria

Tipus	Accés	Scope	Persistència
Global	R/W	GPU i CPU	Programa
Textura	R/W	GPU i CPU	Programa
Constant	R	GPU i CPU	Programa
Shared	R/W	*Threads* d'un bloc	bloc

TAULA 5.1: Resum memòries CUDA

5.3.2 Sincronització

Per permetre la col·laboració de *threads* més enllà de la memòria *shared*, CUDA ens permet coordinar l'execució de *threads* d'un mateix bloc mitjançant la crida `__syncthreads()` dins d'un *kernel*. Aquesta actua com un semàfor que atura els *threads* fins que hagin arribat tots a aquell punt. La sincronització és fonamental en treballar en paral·lel, ja que ens permet accedir de manera segura a recursos compartits, garantir ordre d'execució...

5.4 Problemes de cobertura

Entrant més en detall en el que s'ha explicat al Capítol 1, els problemes de cobertura tracten de col·locar un conjunt limitat de recursos, generalment anomenats serveis o fonts, per tal de cobrir una certa demanda dins d'un entorn. Cada servei segons on estigui col·locat té associada una certa àrea de cobertura, que és el conjunt format per tots els punts que poden arribar a ell amb un camí de longitud inferior o igual al seu radi d'abast.

Un entorn està definit com a qualsevol regió plana d'espai, p. ex. una circumferència, un quadrat, un polígon bidimensional... En aquest projecte ens interessen en particular les regions amb obstacles, conegudes formalment com a no convexes.

Una regió no convexa (Figura 5.5) és aquella en la qual hi ha parelles de punts que no es poden connectar amb un sol segment recte. En canvi, una regió convexa (Figura 5.6) és aquella en la qual totes les possibles parelles de punts sí que poden ser connectades de tal manera.

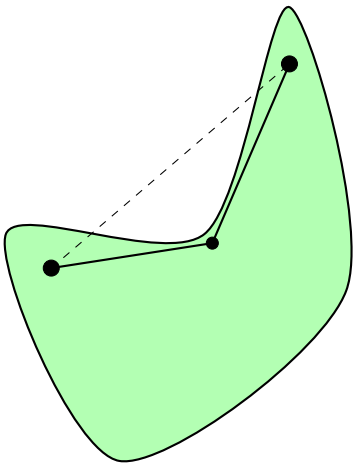


FIGURA 5.5: Exemple de regió no convexa

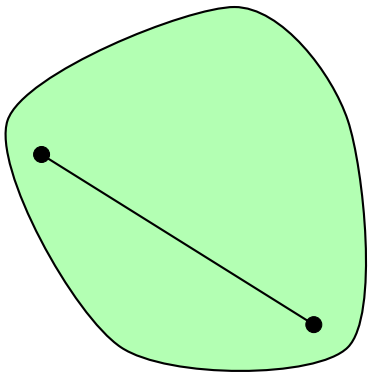


FIGURA 5.6: Exemple de regió convexa

Qualsevol domini amb obstacles, o parets que formin angles obtusos, serà del primer tipus, ja que provocaran que algunes parelles de punts no es puguin connectar amb una sola línia recta.

5.4.1 Problemes de cobertura sense obstacles

En entorns sense obstacles determinar la regió coberta per un servei no té cap complicació, ja que sabem que tots els punts del domini estan connectats amb el servei per una sola línia recta i, per tant, només hem de calcular la distància per discriminar quins queden coberts i quins no.

En aquest cas independentment d'on estigui col·locat un servei la seva regió completa serà una circumferència, total o parcial tallada per la frontera de l'entorn, tal com es pot veure a la figura següent:

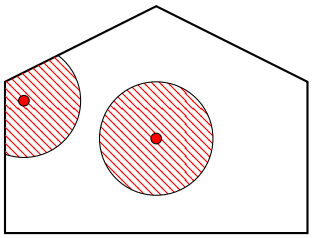


FIGURA 5.7: Cobertura de dos serveis en un entorn poligonal sense obstacles

5.4.2 Problemes de cobertura amb obstacles

En aquests entorns perdem aquesta noció de connexió entre tota parella de punts i, per tant, es necessita un criteri més complex que simplement mirar la distància, ja que aquesta tot i ser un criteri necessari no és suficient: un punt que estigui a una distància major al radi d'abast mai quedarà cobert, però en cas contrari tampoc podem assegurar que ho quedi. Això és causat per la geometria d'entorns amb obstacles, que pot provocar que la distància no sigui sempre igual a la longitud del segment que connecta dos punts, tal com es pot observar a la Figura 5.5, on el camí s'ha de partir en dos segments vorejant la frontera i acaba arribant amb una longitud major a la distància euclidiana (el camí dibuixat és una aproximació, el camí més curt tindria el punt central en contacte amb la frontera). Un altre exemple és el de la Figura 5.8, on hi ha punts no coberts que estan a una distància euclidiana menor o igual que punts coberts. La diferència està en el fet que hi ha obstacles que s'han de vorejar, i vorejar-los incrementa la longitud dels camins que hi arriben i, per tant, la distància del camí més curt més gran que la distància entre ells.

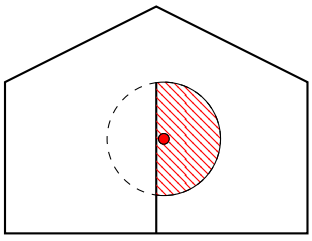


FIGURA 5.8: Cobertura d'un servei en un entorn amb una paret com obstacle

Per evitar possibles ambigüitats, a partir d'ara, quan parlem de distància entre dos punts sempre farem referència a la longitud del camí més curt que els connecta. Per parelles de punts visibles, és a dir sense obstacles pel mig, aquest sempre serà un únic segment recte i per parelles no visibles serà una concatenació de segments

units a les cantonades dels obstacles intermedis (Figura 5.9), que és el punt d'unió que minimitza la longitud.

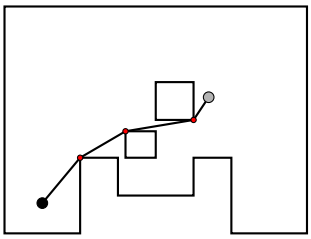
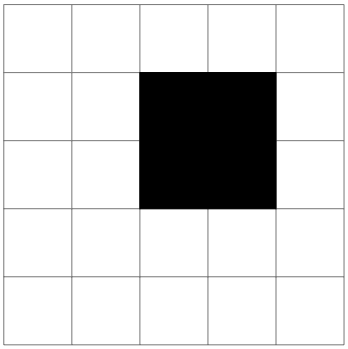


FIGURA 5.9: Concatencació de segments que forma un camí poligonal que evita obstacles intermedis

A la literatura, especialment en el context de *path planning*, existeixen múltiples estratègies per enfocar el problema de trobar camins poligonals, de la menor longitud possible, entre parelles de punts en entorns amb obstacles com per exemple a Crane, Weischedel i Wardetzky 2017 i Valero 2013 que fan servir principis físics i equacions diferencials parcials per trobar-ne. També hi ha estratègies basades en el còmput de transformades de distància, unes operacions matemàtiques que calculen per cada punt d'un domini la menor distància possible a un cert objectiu, com a Elizondo-Leal, Parra-González i Ramírez-Torres 2013, on es presenta un mètode per calcular la *Exact Euclidean Distance Transform* (**EEDT**) (Figura 5.10) mitjançant un esquema de propagació i testos de visibilitat que permet obtenir distàncies exactes entre parelles de punts.



(A) Domini abans de fer l'EEDT

$\sqrt{5}$	$\sqrt{2}$	1	1	$\sqrt{2}$
2	1			1
2	1			1
$\sqrt{5}$	$\sqrt{2}$	1	1	$\sqrt{2}$
$2\sqrt{2}$	$\sqrt{5}$	2	2	$\sqrt{5}$

(B) Domini després de fer l'EEDT

FIGURA 5.10: Exemple d'EEDT d'un domini 5×5 amb un quadrat 2×2 com a objectiu de la transformada

La solució proposada (Capítol 8) es basa en l'obtenció de les distàncies exactes de cada punt fins al servei més proper, de manera anàloga a l'EEDT, utilitzant una variant de l'algoritme de Bellman-Ford (**BF**) com a esquema de propagació. BF és altament versàtil, ja que es pot paral·lelitzar fàcilment i és aplicable a grafs amb cicles de cost negatiu.

A l'Algoritme 1 es pot veure el pseudocodi del cas particular en què el graf no conté cicles de cost negatiu. BF comença inicialitzant la distància al vèrtex que conté el servei a 0 i sobreestimant la de tots els altres vèrtexs del graf inicialitzant-les a infinit. Aquestes aniran disminuint durant el procés de propagació fins a haver aconseguit tots els camins mínims. A cada iteració del BF, cada vèrtex intenta allargar els camins

que arriben als seus veïns fins a ell. A la versió naïf de l'algoritme aquest fa tantes iteracions com arestes té el graf, una versió més eficient itera mentre hi hagi hagut alguna actualització en algun vèrtex del graf.

La principal diferència entre BF i Dijkstra, l'algorisme més eficient conegut de propagació de camins, és que el darrer utilitza una cua de prioritats per propagar el camí que potencialment donarà una longitud més curta a un nou vèrtex. Dijkstra és un algorisme seqüencial, difícilment paral·lelitzable, en canvi, BF analitza tots els vèrtexs a cada iteració i els actualitza tenint en compte només la informació dels vèrtexs adjacents, per tant, és fàcilment paral·lelitzable. La solució que es proposa fa servir una adaptació del BF que propaga distàncies a partir de múltiples seus i que trunca els camins al radi de cobertura dels serveis, se'n veuran els detalls al Capítol 8.

Algorithm 1 Bellman-Ford

```
1:  procedure BellmanFord(Graph G, Source s)
2:      for v ∈ V(G) do
3:          if v ≠ s then
4:              distance[v] = ∞
5:              predecessor[v] = −1
6:          else
7:              distance[v] = 0
8:              predecessor[v] = v
9:          end if
10:     end for
11:     changes = 1
12:     while changes ≡ 1 do
13:         changes = 0
14:         for v ∈ V(G) do
15:             for n ∈ neighbours(v) do
16:                 tentativeDistance = distance[n] + weight(n, v)
17:                 if distance[v] > tentativeDistance then
18:                     distance[v] = tentativeDistance
19:                     predecessor[v] = n
20:                     changes = 1
21:                 end if
22:             end for
23:         end for
24:     end while
25: end procedure
```

5.4.3 Problemes de maximització de cobertura

R.L. Church i C. ReVelle 1974 van formalitzar i introduir el problema de localitzar un nombre específic de serveis per maximitzar la quantitat de demanda coberta, com a regió, dins del seu radi d'abast. El van anomenar *Maximal Covering Location Problem* (**MCLP**) i per resoldre'l consideren dos conjunts d'ubicacions discretes, un representa la demanda i l'altre els possibles emplaçaments dels serveis. En conseqüència, converteixen el **MCLP** en un problema de programació lineal entera que permet algunes variacions. Les variacions provenen d'utilitzar diferents maneres de determinar l'àrea de cobertura d'un servei o assignar pesos a les possibles ubicacions

segons diversos factors i paràmetres. Mentre que la forma estàndard de **MCLP** considera un conjunt finit d'ubicacions candidates per als serveis i un conjunt de punts discrets representen la demanda, la seva generalització, anomenada *Planar MCLP* (R.L. Church 1984), permet que tant els serveis com la demanda es localitzin a qualsevol lloc en el pla continu. N. Megiddo, E. Zemel i S.L. Hakimi 1983 i Hochbaum 1996 van demostrar la naturalesa *NP-Hard* del problema. Per tant, al llarg del temps s'han anat proposant diverses tècniques heurístiques que intenten obtenir solucions acceptables en un temps de còmput raonable.

En el present projecte es tracta el **MCLP** per entorns amb obstacles mitjançant un algoritme genètic.

Optimització amb algoritmes genètics

Els algoritmes genètics són una tècnica d'optimització inspirada en el procés de selecció natural, on les possibles solucions són tractades com a individus d'una població que es poden reproduir i mutar, simulant així el procés d'evolució d'organismes biològics per selecció natural.

El funcionament general dels algoritmes genètics es pot veure a la Figura 5.11. Es comença amb una població inicial d'individus (solucions) generats aleatòriament que entren en el procés d'evolució. Aquest consisteix en tres etapes diferents: *selecció*, on s'utilitza el *fitness* (qualitat de la solució) per escollir quins individus es reproduiran; *reproducció*, on els escollits es recombinen generant dues noves solucions; *mutació*, on amb una probabilitat les noves solucions poden patir una modificació aleatòria. Aquest bucle s'atura quan s'han fet un nombre determinat d'iteracions o quan obtenim una solució prou bona.

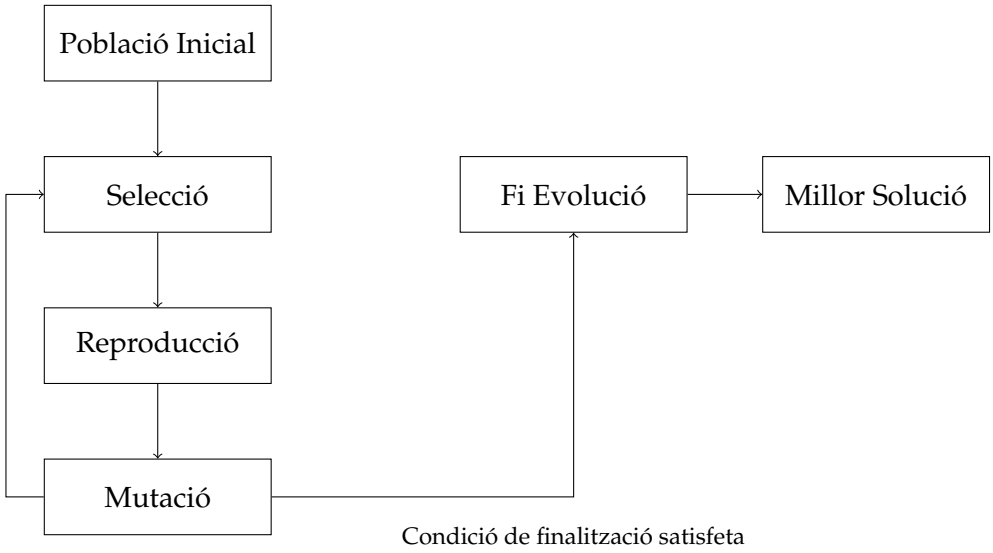


FIGURA 5.11: Diagrama funcionalment algoritme genètic

La selecció consisteix en el procés de tria de parelles reproductores. Dins dels esquemes més utilitzats es troben l'elitista i el de torneig. El primer simplement selecciona els k millors individus de la població. En canvi, el darrer simula un torneig k vegades per escollir els individus, per cada un selecciona n individus aleatoris (mida del torneig), es queda el que tingui major *fitness* i es repeteix el procés. A continuació es mostren ambdós en pseudocodi:

Algorithm 2 Elitist Selection

```
1: procedure ELITISTSELECTION(Population pop, Size k)
2:     sortedPopulation = pop.sort()
3:     nextGeneration = sortedPopulation[1..k]
4:     return nextGeneration
5: end procedure
```

Algorithm 3 Tournament Selection

```
1: procedure TOURNAMENTSELECTION(Population pop, TournamentSize n, Size
   k)
2:     nextGeneration = new List
3:     while length(nextGeneration) < k do
4:         tournament = new List
5:         for i ∈ {1..n} do
6:             randomIndex = rand(1:pop.size())
7:             ind = pop[randomIndex]
8:             tournament.add(ind)
9:         end for
10:        tournament.sort()
11:        nextGeneration.add(tournament.first())
12:    end while
13:    return nextGeneration
14: end procedure
```

Pel que fa a la reproducció, en casos com el d'aquest projecte on els individus es representen mitjançant vectors, normalment consisteix a fer un *one-point crossover* per combinar parelles de vectors. Aquesta tècnica implica combinar parelles de vectors mitjançant la selecció d'un punt d'intersecció. Aquesta pot ser generalitzada a $k-$*point crossover*, on es trien k punts d'intersecció, però només és beneficiosa en casos on es treballa amb vectors extensos.

Algorithm 4 One-Point Crossover

```
1: procedure ONEPOINTCROSSOVER(Individual p1, Individual p2, Index crosso-
   verPoint)
2:     child = new Individual
3:     child = p1[1 : crossoverPoint-1] ++ p2[crossoverPoint : p2.size()]
4:     return child
5: end procedure
```

Finalment, la mutació en vectors consisteix a seleccionar un o més elements (versió k) i actualitzar-los amb valors aleatoris. Aquests generalment estan restringits pels *constraints* del problema.

Algorithm 5 Mutation

```
1: procedure MUTATION(Individual i, Index mutationIndex)
2:     i[mutationIndex] = constraintMin < randomValue() < constraintMax
3: end procedure
```

Capítol 6

Requisits del sistema

Els requisits del projecte que ha de tenir el programa estan dividits en funcionals i no funcionals.

Requisits funcionals del sistema

RF1 Accepta imatges de plànols com a *input*.

RF2 Accepta imatges de polígons com a *input*.

RF3 Calcula mapes de cobertura.

RF4 Troba solucions aproximades del **MCLP**.

RF5 Mostra el temps d'execució.

RF6 Mostra el percentatge de cobertura.

RF7 Mostra el mapa de cobertura calculat.

RF8 Guarda el mapa de cobertura calculat.

Requisits no funcionals del sistema

RNF1 Utilitza algoritmes eficients per calcular mapes de cobertura.

RNF2 Utilitza la GPU per accelerar càlculs.

RNF3 Troba solucions òptimes del **MCLP** en temps raonable.

Requisits funcionals de l'usuari

RF1 Pot configurar nombre de serveis.

RF2 Pot configurar radi d'abast dels serveis.

RF3 Pot entrar manualment una distribució de serveis.

RF4 Pot guardar una representació binària del domini.

RF5 Pot guardar el contingut de les iteracions de **BF** en format `.txt`.

RF6 Pot configurar el nombre de generacions de l'algoritme genètic.

RF7 Pot configurar la mida de la població de l'algoritme genètic.

RF8 Pot configurar la probabilitat de mutació de l'algoritme genètic.

RF9 Pot configurar la condició de finalització de l'algoritme genètic.

Requisits no funcionals de l'usuari

RNF2 Utilitza la interfície gràfica per interactuar amb el sistema.

Cal tenir en compte que els requisits funcionals de l'usuari que s'han especificat estan pensats per un context de recerca, on aquest està familiaritzat amb els detalls tècnics de la implementació.

En el cas que l'aplicació final fos destinada a usuaris d'un públic més general funcionalitats com RF4 i RF5 desapareixerien, ja que estan pensades per analitzar el comportament de l'algoritme. També les opcions de configuració de l'algoritme genètic, RF6-RF9, serien reemplaçades per opcions més intuïtives com per exemple: cerca bàsica, cerca moderada i cerca exhaustiva. Que farien servir uns certs paràmetres preestablerts, simplificant així l'ús del programa.

Capítol 7

Estudis i decisions d'implementació

7.1 Llibreries utilitzades

Les llibreries que es mencionen a continuació s'han utilitzat per: programació de la GPU, gestió i tractament d'imatges, desenvolupament de la interfície gràfica i disseny de gràfics i diagrames de la memòria, respectivament.

7.1.1 CUDA

La llibreria CUDA i el CUDA Toolkit proporcionen una plataforma de còmput paral·lel per les targetes gràfiques de la marca NVIDIA. La llibreria inclou una sèrie de funcions optimitzades per la GPU, des de la gestió de memòria fins a operacions d'àlgebra lineal i transformades de Fourier. El toolkit consisteix en un conjunt de compiladors i eines que faciliten el desenvolupament. La llicència, NVIDIA Software License Agreement (SLA), permet un ús no comercial i acadèmic de manera gratuïta. Un dels principals beneficis d'usar aquesta llibreria és la senzilla intolerabilitat que permet entre codi de la CPU i el codi de la GPU.

Com que CUDA ja s'havia fet servir en anteriors projectes del departament es va decidir continuar-ne l'ús en aquest.

7.1.2 OpenCV

OpenCV, Open Source Computer Vision, és una popular llibreria de visió per computador que conté un gran conjunt de funcions relacionades amb el tractament d'imatges i vídeos. La seva llicència és Apache 2.0 i, per tant, permet tant ús com modificació gratuïta pels casos comercial i no comercial.

Tot i existir alternatives a OpenCV, com SimpleCV o scikit-image, es va escollir aquesta perquè ja estava familiaritzat amb ella d'assignatures del grau com informàtica industrial i robòtica, on s'utilitza per fer les activitats pràctiques. Un altre motiu important a l'hora d'escollir-la va ser la seva documentació, ja que és excel·lent i conté molts exemples que faciliten el desenvolupament.

7.1.3 WinForms

La llibreria WinForms, abreviació de Windows Forms, és un *framework* de Microsoft que permet la creació d'interfícies gràfiques a la plataforma Windows de manera senzilla i efectiva. Forma part del `.NET` Framework i comparteix la mateixa llicència MS-PL, Microsoft Public License, que permet el seu ús, modificació i redistribució de manera gratuïta de forma comercial i no comercial.

La simplicitat va ser el principal motiu per escollir aquesta llibreria. Winforms suporta *drag and drop* per la construcció d'interfícies i una senzilla API per afegir funcionalitat als elements de la interfície que permet fer dissenys funcionals de manera ràpida i efectiva. Alternatives com OpenGL, tot i ser més potents també són més complicades d'utilitzar, i com que l'objectiu principal del projecte no era fer una interfície complexa es va decidir utilitzar Winforms.

7.1.4 TikZ

TikZ és una llibreria de LaTeX que permet la creació de gràfics professionals en format vectorial amb un alt nivell de control sobre els detalls. En aquest projecte s'ha fet servir per fer la gran majoria de representacions gràfiques que hi ha presents a la memòria, incloent-hi els diagrames UML.

7.2 Programari utilitzat

7.2.1 Visual Studio

Visual Studio és un entorn de programació creat per Microsoft que proporciona un conjunt complet d'eines que facilita el desenvolupament per un gran nombre de llenguatges. Per programar en CUDA és l'entorn més fet servir, ja que té una integració nativa i documentació de primera classe. Tot i existir alternatives, Visual Studio es considera l'estàndard per programar GPU en CUDA, pel fet que el seu suport és molt superior a la resta.

7.2.2 GitHub

GitHub és una plataforma de control de versió de programari que funciona amb el sistema Git, que permet gestionar i seguir els canvis en el codi font dels projectes. A través de la plataforma es poden crear i emmagatzemar repositoris que són com carpetes corresponents a projectes.

Aquesta plataforma s'introdueix a l'assignatura projecte de desenvolupament de software i, per tant, ja estava familiaritzat amb el seu ús i com fer-lo servir en un projecte d'aquest estil.

7.2.3 Overleaf

Overleaf és un editor *online* de documents en format LaTeX que permet als seus usuaris escriure, editar i col·laborar en documents de manera gratuïta. És àmpliament utilitzat per l'edició de textos en l'àmbit cientificotècnic, ja que ofereix un entorn de treball senzill que permet la creació de documents professionals sense haver d'instal·lar cap software. En el marc d'aquest projecte s'ha emprat per elaborar la memòria, seguint la plantilla proporcionada.

Capítol 8

Anàlisi i disseny de la solució

En aquest capítol es presenta l'anàlisi i el disseny del mètode de càlcul de mapes de cobertura. Aquest càlcul es basa en un procés iteratiu en el qual col·laboren la GPU i la CPU repartint-se tasques paral·leles, seqüencials i de sincronització.

El procés d'obtenció del mapa de cobertura d'un conjunt de serveis rep com a entrada una imatge, que representa el domini que es vol cobrir, el radi d'abast dels serveis i la seva distribució. El domini s'obté extraient la frontera i els obstacles de la imatge mitjançant un mètode conegut com a descomposició en cel·les. Aquesta informació es guarda en una matriu on cada punt, píxel de la imatge o element de la matriu, està classificat com a espai lliure o ocupat. La distribució dels serveis es llegeix de la interfície d'usuari o es genera de manera aleatòria a partir del nombre de serveis que es volen col·locar.

El resultat és el mapa de cobertura corresponent, representat com una matriu de les mateixes dimensions que el domini, on cada punt està classificat com a punt cobert o no cobert i el percentatge d'àrea coberta respecte al total de punts lliures del domini.

Aquest resultat s'utilitza llavors com a funció objectiu de l'algorisme genètic. És a dir, el percentatge de punts coberts sobre punts interiors associat a una distribució de serveis es tracta com la funció a optimitzar.

8.1 Obtenció del domini

La primera etapa de la solució consisteix en l'obtenció i inicialització del domini a partir d'una imatge. Aquesta representa un entorn com un plànol o un polígon, on les parets i els obstacles es troben marcats amb un color fosc proper al negre.

A partir d'aquesta imatge, s'obté una matriu de les mateixes dimensions que serveix com a representació del domini, on cada cel·la indica si el corresponent píxel de la imatge és lliure o està ocupat per un obstacle. Aquest procés és conegut com a descomposició en cel·les i a la Figura 8.1 se'n pot veure un exemple.

Tenint en compte que el domini és constant, aquest es guarda en una textura CUDA per aprofitar els beneficis de la memòria de textures de la GPU. Això permet realitzar consultes ràpides als veïns durant l'algoritme de propagació, ja que la memòria cau d'aquesta està adaptada per a consultes d'aquest estil, on hi ha una alta localitat espacial.

Amb l'objectiu d'evitar càlculs addicionals durant l'etapa posterior de correcció de distàncies, s'identifiquen no només quins píxels estan lliures i quins estan ocupats, sinó també les cantonades dels obstacles, definides en detall al primer apartat del Capítol 10. Aquesta informació és important per delimitar els segments dels camins.

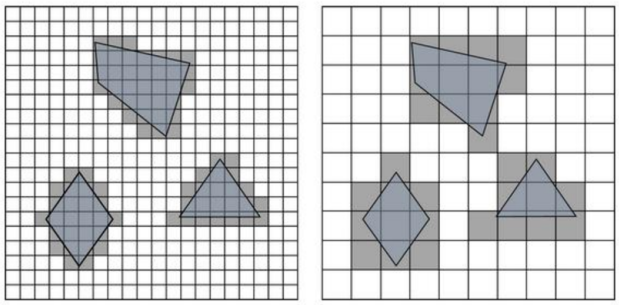


FIGURA 8.1: Exemple descomposició en cel·les d'un domini amb obstacles

La descomposició en cel·les es du a terme a la GPU. S'assigna un *thread* a cada píxel de la imatge que fa les operacions necessàries per determinar si aquest és lliure, ocupat o cantonada.

8.2 Càlcul de cobertura amb expansió pseudoeuclidiana

La solució proposada inicia el càlcul de mapes de cobertura amb l'adaptació de l'algoritme de Bellman-Ford (Algoritme 6) en paral·lel que tracta el domini com un graf.

A aquesta, cada cel·la determina el cost d'actualitzar la seva informació a partir d'una finestra de consulta, representada a la Figura 8.2, amb mètrica pseudoeuclidiana, que juga el mateix rol que les arestes en un graf amb pesos, és a dir determina el preu d'anar d'un node a un seu veí, i serveix conjuntament amb la matriu del domini com a guia de l'esquema de propagació, permetent només propagar cap a cel·les lliures d'obstacles.

Els resultats intermedis d'aquest procés es van anotant en una matriu de distàncies, de les mateixes dimensions que el domini, que acabarà representant el mapa de cobertura un cop finalitzada la propagació.

$\sqrt{2}$	1	$\sqrt{2}$
1	0	1
$\sqrt{2}$	1	$\sqrt{2}$

FIGURA 8.2: Finestra de consulta amb mètrica pseudo-euclidiana

Aquesta propagació es pot veure esquematitzada a la Figura 8.3 per a un domini de dimensions 7×7 sense obstacles, on un servei està situat al centre. A cada iteració, es marquen en negre els punts coberts i en gris els veïns no coberts dels punts coberts, els quals tindran l'oportunitat d'actualitzar-se per primera vegada a la següent iteració.

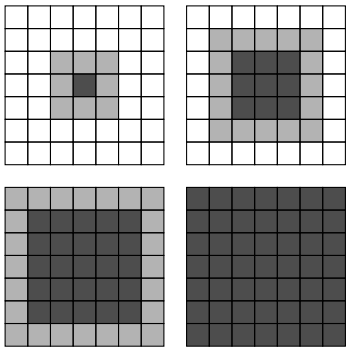


FIGURA 8.3: Propagació dins d'un domini 7x7 sense obstacles.

8.2.1 Adaptació de l'algoritme de Bellman-Ford en paral·lel

A continuació es presenta l'adaptació de l'algoritme de Bellman-Ford utilitzada per a realitzar el càlcul de cobertura. A diferència de la versió seqüencial en aquesta versió cada píxel de la imatge és processat de manera independent per un *thread* que consulta els costos dels camins que venen dels veïns a partir de la finestra de consulta. En aquesta implementació, no es fa ús dels predecessors, ja que aquests no influeixen en la propagació de distàncies.

Algorithm 6 Parallel-Bellman-Ford

```
1:  procedure ParallelBellmanFord(Domain 𝒟, Sources S, Range r)
2:      for p ∈ 𝒟 do (in parallel)
3:          if p ∉ S then
4:              distance[p] = r + ε
5:          else
6:              distance[p] = 0
7:          end if
8:      end for
9:      changes = 1
10:     while changes ≡ 1 do
11:         changes = 0
12:         for p ∈ D do (in parallel)
13:             for n ∈ neighbours(p) do
14:                 tentativeDistance = distance[n] + cost(n, p)
15:                 if D[n] ≠ −1 and distance[p] > tentativeDistance then
16:                     distance[p] = tentativeDistance
17:                     changes = 1
18:                 end if
19:             end for
20:         end for
21:     end while
22: end procedure
```

Els neighbours(p) són els píxels no ocupats presents a la finestra de consulta centrada al punt p, representada per la crida cost(n, p).

Errors d'aproximació

Malgrat que les distàncies comunicades als veïns siguin precises en l'àmbit local, en global s'obtenen resultats incorrectes, com es pot apreciar a la Figura 8.4. Això es deu a la naturalesa de l'esquema de propagació utilitzat, el qual només garanteix distàncies exactes per a rutes estrictament horitzontals, verticals i diagonals, tal com es mostra a la Figura 8.5a.

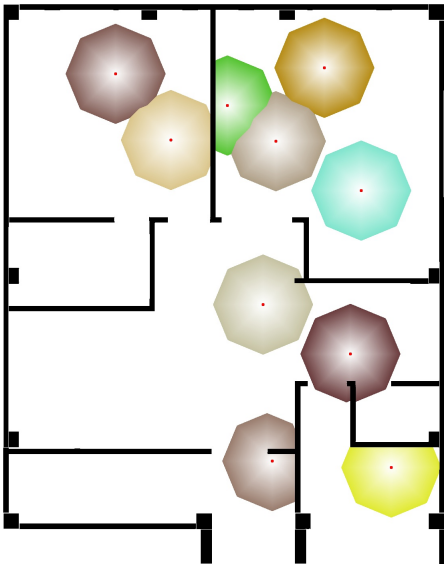


FIGURA 8.4: Mapa de cobertura obtingut per expansió pseudoeuclidana en un entorn amb 10 serveis.

En el cas dels punts que no es troben alineats amb un servei en cap de les direccions mencionades, els camins resultants que els connecten consisteixen en una combinació de segments diagonals i rectes, semblant a utilitzar la mètrica de Manhattan afegint-hi les diagonals. Aquesta combinació de segments difereix significativament en longitud respecte a la distància euclidiana exacta, com es pot observar a la Figura 8.5b.

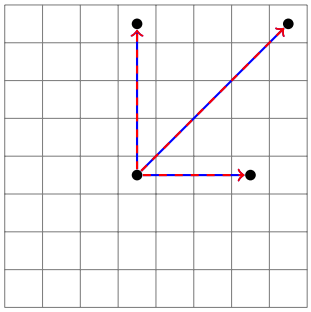
Aquesta situació provoca que, tot i que l'algoritme propaga distàncies exactes a escala local, els resultats globals siguin només parcialment correctes. Tal com es pot veure a la Figura 8.5b els camins representats a la Figura 8.5a són alhora camins òptims i propagats per la matriu, mentre que a la Figura 8.5b el camí vermell no s'obtindrà per propagació, la distància del punt Inici al Fi serà sobreestimada per la longitud del camí representat en blau.

Per tal d'obtenir regions de cobertura més precises, es va decidir afegir una nova etapa a l'algoritme per corregir les distàncies propagades.

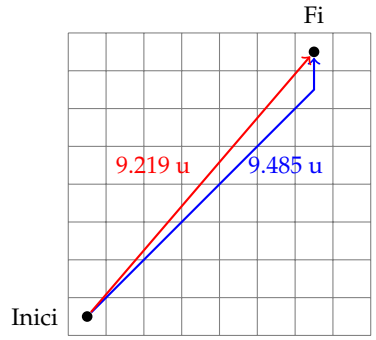
8.2.2 Correcció de distàncies

Amb l'objectiu de corregir els errors d'aproximació de l'esquema de propagació anterior, s'incorpora , en una mena de post-procés, una fase de correcció de distàncies basada en el còmput de la distància euclidiana (Equació 8.1) dels segments que formen els camins.

Per poder calcular les distàncies dels camins, respectant els obstacles, és estrictament necessari poder reconstruir el camí i, per tant, cal saber per on passa. En arribar a un



(A) Camí obtingut per propagació (blau) que coincideix en longitud i recorregut amb l'òptim (vermell)



(B) Camí obtingut per propagació (blau) no coincideix en longitud ni recorregut amb l'òptim (vermell)

FIGURA 8.5: Diferències entre el que s'obté per propagació i el cas òptim, és a dir el que utlilitza distància euclidiana

punt del domini cal saber si el camí ve directament d'un servei o d'una cantonada (això indica que ha vorejat un obstacle). Per això és necessari guardar informació dels predecessors, que definiran els vèrtexs dels camins poligonals.

Aquesta nova fase de correcció de distàncies s'aplica als camins trobats durant la propagació de les distàncies. Mitjançant la substitució de les distàncies aproximades per les distàncies exactes, s'espera minimitzar els possibles errors d'aproximació, mantenint alhora el mateix esquema de propagació, restringit a l'espai lliure del domini, utilitzat anteriorment.

$$d(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{8.1}$$

Per a un camí $\mathcal{C}$ format per un conjunt de segments S_i, amb punts inicials i finals denotats com p_0 i p_n respectivament, la seva longitud es calcula com la suma de les longituds dels seus segments:

$$||\mathcal{C}||_2 = \sum_i ||S_i||_2 \quad \text{on} \quad ||S_i||_2 = d(p_{i_0}, p_{i_n}) \tag{8.2}$$

A la Figura 8.6 es pot veure un exemple que il·lustra la separació d'un camí en dos segments mitjançant un predecessor. Amb l'equació anterior (Equació 8.2) és possible calcular la longitud exacta d'aquest camí de manera senzilla (Equació 8.3).

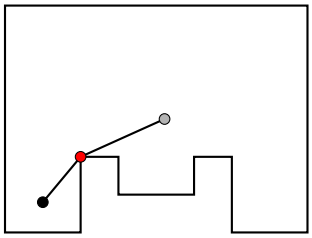


FIGURA 8.6: Camins format per dos segments.

$$||\mathcal{C}||_2 = \sum_{i}^{2} ||S_i||_2 = ||S_0||_2 + ||S_1||_2 = d(p_{ini}, p_{pred}) + d(p_{pred}, p_{cob}) \tag{8.3}$$

Tal com es pot observar en aquest exemple, els punts predecessors actuen com a límits entre els diferents segments que conformen un camí. Això implica que aquests només poden ser cantonades, és a dir, punts on hi ha un canvi de direcció i el camí passa de ser una línia recta a una poligonal.

Aquests predecessors seran serveis o cantonades, marcades durant el procés d'obtenció del domini, ja que són els únics punts que poden actuar com a vèrtexs d'un camí poligonal que minimitzi la longitud entre dos punts.

Durant el procés de propagació del **BF** cal anar guardant els predecessors, per tal de llavors poder recuperar el camí que ens ha portat fins al punt en qüestió.

A través d'aquesta divisió en segments a partir de predecessors, es pot calcular amb precisió la longitud de cada secció d'un camí, millorant la precisió dels resultats obtinguts.

Obtenció dels camins més curts no restringits al graf

En aquesta versió actualitzada de l'algoritme, cada cel·la del mapa de cobertura emmagatzema tant la distància com el predecessor associat al camí que li proporciona la distància. Quan un píxel rep una distància menor a la qual té guardada estenent un camí provinent d'un veí, s'actualitza la distància associada al píxel i s'assigna el predecessor corresponent. Si el veí és una cantonada, el mateix veí és assignat com a predecessor. En cas contrari, s'assigna el predecessor del veí.

Completada la fase de propagació amb els predecessors, tenint en compte els errors d'aproximació, l'algoritme procedeix calculant les distàncies exactes dels camins trobats, com es mostra a l'exemple de l'Equació 8.3.

Un cop s'ha dut a terme aquest càlcul, es pot fer una nova ronda d'expansió, sobre el graf, ja que la majoria dels punts han estat actualitzats amb una distància menor a la qual tenien guardada.

Aquesta parella de processos d'expansió, sobre el graf, i correcció, fora del graf, es repeteix iterativament fins que totes les distàncies i camins convergeixen, obtenint un resultat similar al que es pot observar a la Figura 8.7.

A l'Algoritme 7 es detalla el procediment paral·lel de correcció de les distàncies i a l'Algoritme 8 es poden veure reflectits els canvis corresponents en l'esquema de propagació. A través d'aquestes modificacions, tal com es pot apreciar a la figura anterior, l'algoritme millora la precisió dels resultats considerablement.

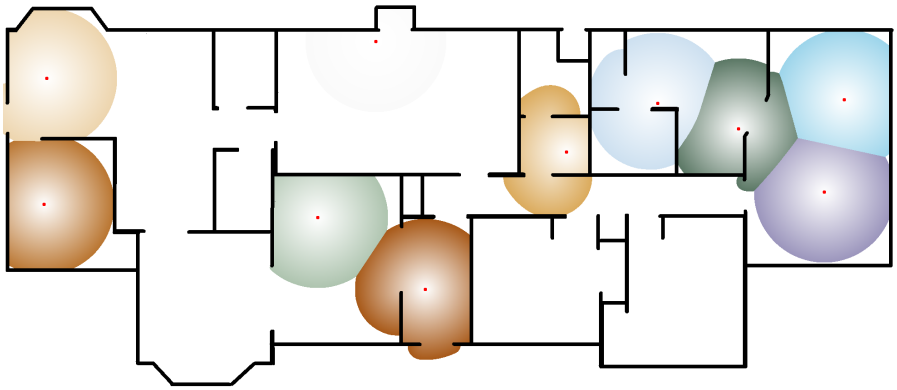


FIGURA 8.7: Mapa de cobertura obtingut per expansió pseudoeuclidiana amb correcció de distàncies en un entorn amb 10 serveis

Algorithm 7 Distance Correction

```
1:  procedure DISTANCECORRECTION(Domain 𝒟)
2:      for p ∈ 𝒟 do (in parallel)
3:          distance = 0.0
4:          currentPixel = p
5:          nextGoal = predecessor[p]
6:          while nextGoal ≠ currentPixel do
7:              distance += euclideanDistance(currentPoint, nextGoal)
8:              currentPoint = nextGoal
9:              nextGoal = predecessor[currentPoint]
10:         end while
11:     end for
12: end procedure
```

Algorithm 8 Parallel-Bellman-Ford

```
1: procedure ParallelBellmanFord(Domain 𝒟, Sources S, Range r)
2:     for p ∈ 𝒟 do (in parallel)
3:         if p ∉ S then
4:             distance[p] = r + ε
5:             predecessor[p] = −1
6:         else
7:             distance[p] = 0
8:             predecessor[p] = p
9:         end if
10:    end for
11:    changes = 1
12:    while changes ≡ 1 do
13:        changes = 0
14:        for p ∈ D do (in parallel)
15:            for n ∈ neighbours(v) do
16:                tentativeDistance = distance[n] + cost(n, p)
17:                if D[n] ≠ −1 and distance[p] > tentativeDistance then
18:                    distance[p] = tentativeDistance
19:                    if isCorner(n) then
20:                        predecessor[p] = n
21:                    else
22:                        predecessor[p] = predecessor[n]
23:                    end if
24:                    changes = 1
25:                end if
26:            end for
27:        end for
28:    end while
29: end procedure
```

Problemes de visibilitat

Les modificacions introduïdes a la secció d'obtenció de camins no restringits al graf, tot i que milloren els resultats considerablement, plantegen una nova complicació: l'aparició de propagacions il·legals de predecessors, que provoquen la formació de camins que travessen obstacles.

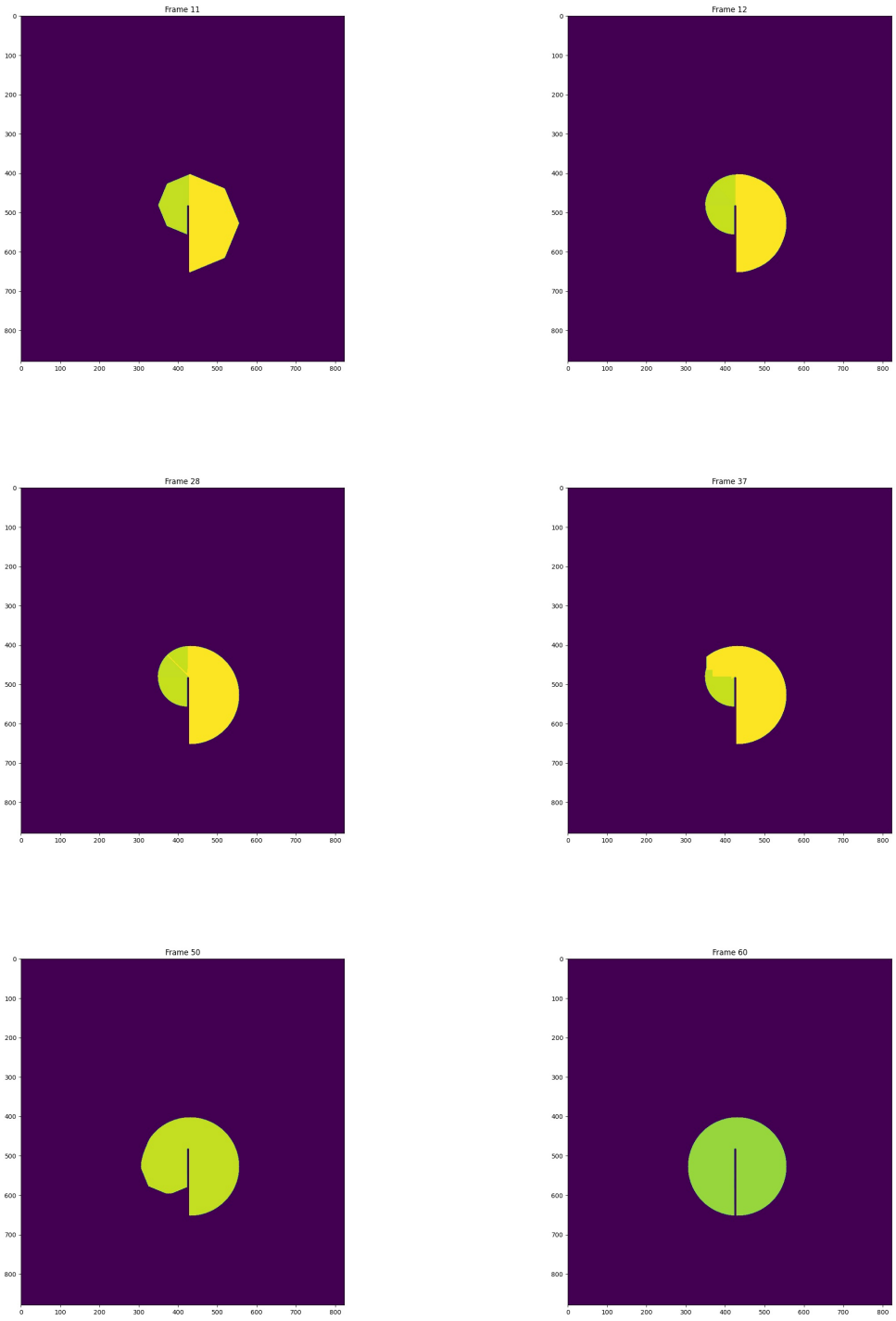


FIGURA 8.8: Representació gràfica dels predecessors d'un mapa de cobertura on falla la propagació

Aquesta problemàtica només apareix en un nombre molt reduït d'instàncies, però impossibilita el tractament del **MCLP**. Això és degut al fet que si tan sols una solució de la població de l'algoritme genètic presenta aquest comportament, molt probablement acabarà guanyant, ja que el seu fitness serà superior a solucions vàlides que sí que respectin obstacles. A la Figura 8.8 es pot veure un exemple, detallat per iteracions, on la propagació de predecessors falla i s'acaba obtenint una solució no vàlida amb més cobertura que la que hauria de tenir.

La causa d'aquest mal comportament és que hi ha camins equivalents que no tots són vàlids. A la Figura 8.9 es poden observar dos camins de la mateixa longitud que comuniquen diferents predecessors al píxel objectiu. El camí vermell li comunica el servei, ja que no ha passat per cap cantonada, en canvi, el camí negre li comunica la cantonada per la qual ha passat. El problema està en el fet que tot i que els dos camins són equivalents pel que fa a la longitud només un d'ells comunica un predecessor vàlid. Sense cap criteri addicional, el punt objectiu s'apuntarà el camí que li arribi primer. A més a més, com que la propagació es fa en paral·lel no hi ha cap manera de controlar quin camí arribarà primer.

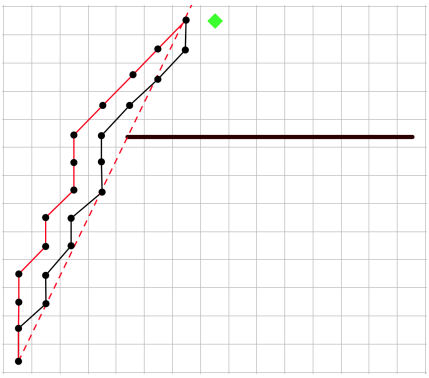


FIGURA 8.9: Exemple de dos camins equivalents que comuniquen diferents predecessors. La línia discontinua marca el límit de visibilitat del servei.

Malgrat els esforços realitzats durant el desenvolupament del projecte, no s'ha pogut resoldre el problema de manera satisfactòria i, per tant, queda com a treball futur. No obstant això, per fer tractable el **MCLP** s'han dissenyat i implementat tres mecanismes diferents amb l'objectiu d'evitar aquestes propagacions il·legals de predecessors. Aquests mecanismes es presenten com a solucions temporals per mitigar aquesta problemàtica i evitar la generació de camins no vàlids.

El primer d'aquests mecanismes utilitza l'argument geomètric que es mostra la Figura 8.10. En cas de la figura el píxel p queda a la dreta del vector $\overrightarrow{p_2p_1}$. Llavors, només els píxels veïns n que quedin a la dreta del vector $\overrightarrow{p_2p_1}$ i a la dreta del vector $\vec{p_1p}$ poden propagar la distància cap a p. De manera similar, quan p queda a l'esquerra del vector $\overrightarrow{p_2p_1}$, només els píxels veïns n que queden a l'esquerra del vector $\vec{p_2p_1}$ i a l'esquerra del vector $\overrightarrow{p_1p}$ poden propagar la distància cap a p. Cal dir que aquesta restricció s'aplica només píxels veïns n que coberts per la mateixa font que cobreix p.

El segon mecanisme consisteix en el fet que quan un punt rep informació d'un veí abans d'acceptar-la o descartar-la comprovarà si és similar a la que té guardada. Si aquesta és de prou similar, de l'ordre de $1e{-}5$ per tenir en compte possibles errors

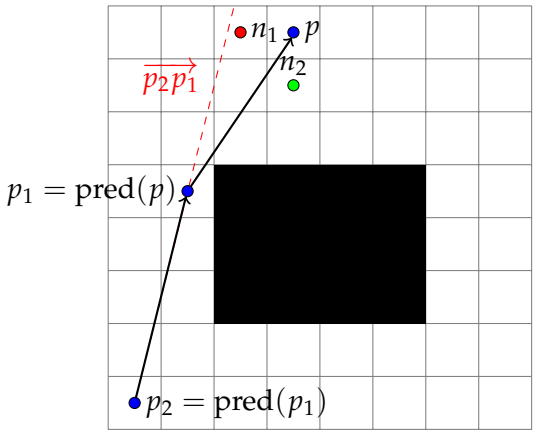


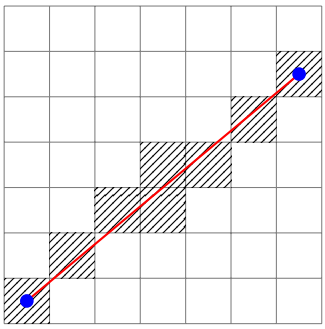
FIGURA 8.10: Píxels veïns que no poden propagar distàncies (vermell) i píxels que si poden propagar distàncies (verd)

de precisió, llavors optarà per quedar-se amb la proposta que li doni el predecessor més proper.

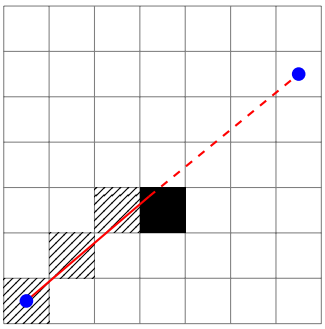
Casos com els de la Figura 8.9 se solucionen amb aquests dos mecanismes, però no solucionen tots els errors.

Donat que cap dels dos mecanismes anteriors proporciona resultats perfectes, tant de forma individual com conjunta, s'introdueix un tercer i final mecanisme: el test de visibilitat. Aquest, malgrat ser el més costós en termes computacionals, soluciona tots els problemes d'aquest tipus i ens permet descartar totes les propagacions incorrectes. A l'Algoritme 10 es pot veure com s'incorpora aquest a l'algoritme de propagació.

Aquest test consisteix a determinar, comprovant manualment, si el predecessor que es comunica a un punt és visible des d'aquest. Per fer aquesta comprovació s'utilitza l'algoritme de Bresenham (Algoritme 9) modificat per detectar obstacles. A la Figura 8.11 es pot veure exemplificat aquest procés per dos punts visibles i no visibles.



(A) Cas on el test de visibilitat funciona



(B) Cas on el test de visibilitat falla.

FIGURA 8.11: Representació del test de visibilitat per un cas visible i un no visible

Algorithm 9 Bresenham's line algorithm

```
1: procedure VisTest(Domain 𝒟, Coordinate x0, Coordinate y0, Coordinate x1, Coordinate y1)
2:     dx = |x1 − x0|
3:     dy = |y1 − y0|
4:     sx = if (x0 < x1) then 1 else −1
5:     sy = if (y0 < y1) then 1 else −1
6:     error = dx − dy
7:     while true do
8:         if isWall(𝒟, x0, y0) then
9:             return false
10:        end if
11:        if x0 = x1 and y0 = y1 then
12:            return true
13:        end if
14:        e2 = 2 × error
15:        if e2 > −dy then
16:            error += dy
17:            x0 += sx
18:        end if
19:        if e2 < dx then
20:            error += dx
21:            y0 += sy
22:        end if
23:    end while
24: end procedure
```

Algorithm 10 Parallel-Bellman-Ford

```
1:  procedure ParallelBellmanFord(Domain 𝒟, Sources S, Range r)
2:      for p ∈ 𝒟 do (in parallel)
3:          if p ∉ S then
4:              distance[p] = r + ε
5:              predecessor[p] = −1
6:          else
7:              distance[p] = 0
8:              predecessor[p] = p
9:          end if
10:     end for
11:     changes = 1
12:     while changes ≡ 1 do
13:         changes = 0
14:         for p ∈ D do (in parallel)
15:             for n ∈ neighbours(v) do
16:                 tentativeDistance = distance[n] + cost(n, p)
17:                 if D[n] ≠ −1 and distance[p] > tentativeDistance then
18:                     distance[p] = tentativeDistance
19:                     neighPredVisible = visTest(p, predecessor[n])
20:                     if isCorner(n) || !neighPredVisible then
21:                         predecessor[p] = n
22:                     else
23:                         predecessor[p] = predecessor[n]
24:                     end if
25:                     changes = 1
26:                 end if
27:             end for
28:         end for
29:     end while
30: end procedure
```

8.3 Cerca de la cobertura màxima

La solució proposada empra un algoritme genètic per tractar el **MCLP**, utilitzant el percentatge de cobertura com a funció objectiu per a l'optimització. En aquest context, els elements principals de l'algoritme genètic prenen la següent forma:

Concepte	Equivalent a MCLP
Individu	Distribució de serveis
Reproducció	Creació d'una nova distribució a partir la combinació de dos altres
Mutació	Pertorbació d'una coordenada d'un servei d'una distribució
Fitness	Percentatge de cobertura d'una determinada distribució

TAULA 8.1: Equivalència de conceptes

Les solucions candidates, individus, es representen com a vectors d'enters de coordenades, on els elements parells representen les coordenades x i els imparells representen les coordenades y. Aquesta representació permet emmagatzemar eficientment un gran nombre de solucions i delegar tots els càlculs intensius en la GPU, evitant transferències redundants de dades. A més, simplifica considerablement la implementació dels operadors genètics, com s'ha pogut observar al Capítol 5.

Les tasques queden repartides entre la CPU i GPU de la següent manera:

CPU	GPU
Càlcul de selecció	Càlcul del fitness
Càlcul de reproducció	Gestió del domini
Càlcul de mutació	Gestió mapes de cobertura
Gestió de població	

FIGURA 8.12: Repartiment de tasques entre la GPU i la CPU

La CPU és responsable de les tasques seqüencials, mentre que la GPU s'encarrega de les tasques paral·leles. A més, la GPU gestiona el domini i els mapes de cobertura per tal de minimitzar les transferències de dades innecessàries. Tant el domini com els mapes de cobertura s'envien a la GPU abans d'iniciar l'algoritme genètic i es mantenen allà durant tota l'execució. Només es recupera de la GPU un únic mapa de cobertura, el de la millor solució obtinguda, un cop finalitzat el bucle d'evolució.

Pel que fa als mapes de cobertura, la GPU en guarda només un que es reinicialitza abans de realitzar els càlculs necessaris per a cada candidat.

Per evitar la repetició dels càlculs més costosos per a cada solució candidata, aquestes es guarden en forma de parella amb el seu *fitness*, que inicialment es posa a zero. Això permet evitar el càlcul del mapa de cobertura de nou en cas que aquesta solució sigui seleccionada per una generació futura.

Els paràmetres per defecte seleccionats a l'algoritme estan basats en les recomanacions de Poli et al. 2008, on es menciona que, en general, és més beneficiós tenir una

població gran i poques generacions que al revés. Pel disseny dels operadors genètics, que es detallen al Capítol 10, també s'han tingut en compte les indicacions de Koza 1992

Capítol 9

Anàlisi i disseny del sistema

9.1 Disseny de la interfície gràfica

Tal com s'ha indicat al Capítol 6, els requisits del sistema han estat pensats per a ser utilitzats en un context d'investigació, per la qual cosa es dona una especial importància en tenir una interfície que permeti un control detallat de l'algoritme. Tenint en compte aquesta consideració, el diagrama de casos d'ús d'un usuari pren la següent forma:

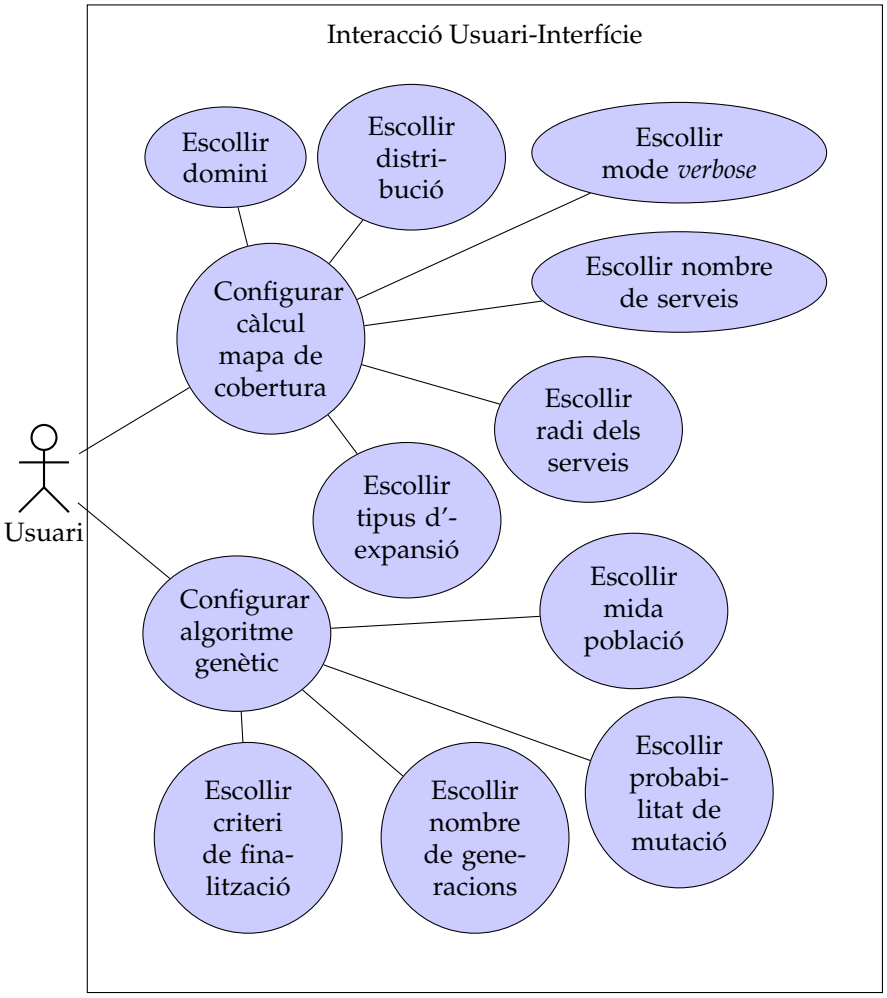


FIGURA 9.1: Diagrama cas d'ús d'usuari.

Cada una de les operacions representades en el diagrama de cas d'ús (Figura 9.1) correspon a un dels requisits funcionals de l'usuari, especificats al Capítol 6.

Aquestes operacions s'agrupen de la mateixa manera que es mostra al diagrama a la interfície gràfica, com es pot observar en el disseny inicial representat a la Figura 9.2.

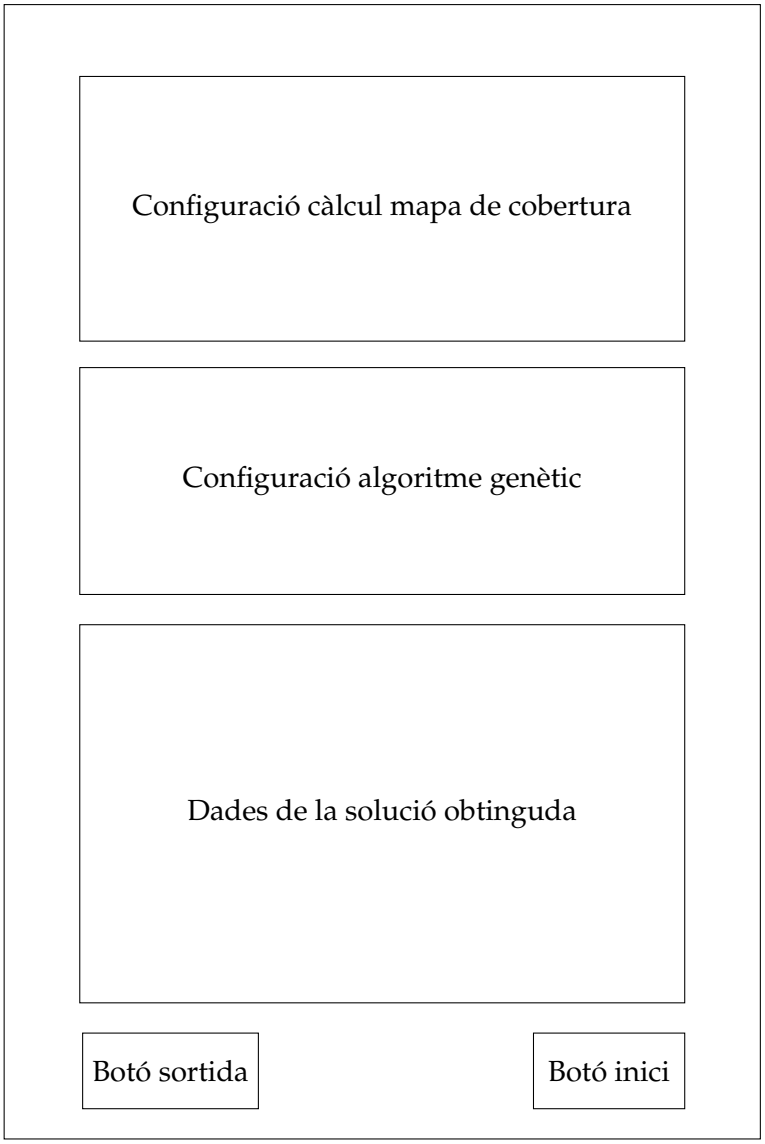


FIGURA 9.2: Disseny inicial de la interfície gràfica

9.2 Model de dades

El codi del projecte està organitzat principalment en *namespaces*, que engloben funcionalitats específiques relacionades, i *structs*, que agrupen conjunts de paràmetres. Aquest enfocament permet tenir una organització clara i modular del codi sense incloure classes quan no són necessàries. Els únics casos on s'ha considerat adequat crear-ne en comptes d'un *namespace* o *struct* ha sigut en els casos del *solver* (algoritme genètic) i de les cel·les del mapa de cobertura, tal com es podrà veure a la següent secció.

Els fitxers del programa estan organitzats d'acord amb el que es mostra a la Figura 9.3.

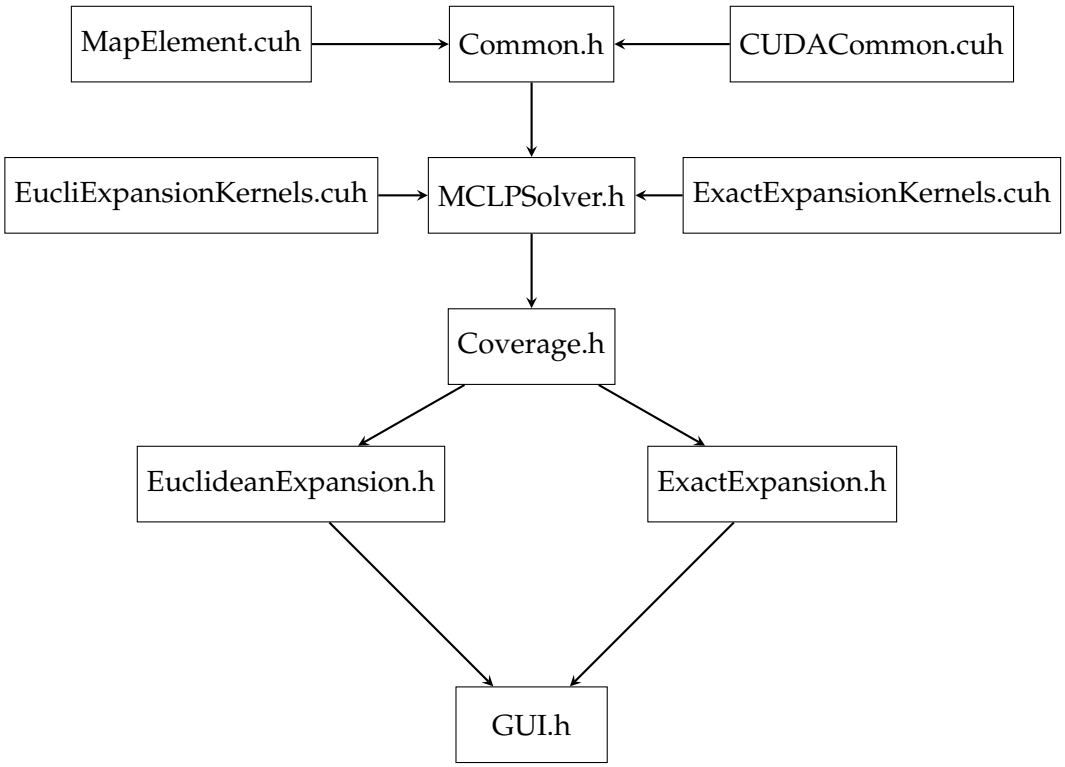


FIGURA 9.3: Diagrama de dependències

Abans d'entrar en detall respecte al contingut de cada un dels fitxers, fem èmfasis en l'estructura jeràrquica que els organitza:

En el nivell més alt, trobem els que contenen la base dels algoritmes, `MapElement.cuh`, `Common.h` i `CUDACommon.cuh`.

En el següent nivell de l'estructura, trobem els fitxers que contenen la implementació de la solució proposada, `EucliExpansionKernels.cuh`, `ExactExpansionKernels.cuh` i `MCLPSolver.cuh`.

Finalment, trobem els fitxers `Coverage.h`, que conté crides a alt nivell de la solució proposada, i `EuclideanExpansion.h` i `ExactExpansion.h`, que contenen el codi que gestiona la interacció entre la interfície, `GUI.h`, i els algoritmes.

9.2.1 Diagrames de classe

Seguint el mateix ordre, primer tenim la classe `MapElement`, Figura 9.4, que representa una cel·la del mapa de cobertura i està implementada per poder ser utilitzada tant a la GPU com a la CPU. Com a atributs té la distància, l'índex del predecessor i el del servei més proper. Aquesta es fa servir en format d'*array* per representar els mapes de cobertura.

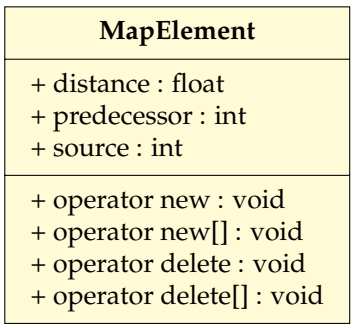


FIGURA 9.4: Diagrama de classe de `MapElement`

Dins del fitxer `Common.h` tenim els *namespaces*:

- **UTILS** (Figura 9.5): conté funcions que s'utilitzen a diversos punts del programa, però no encaixen a cap punt específic.
- **IO** (Figura 9.6): conté totes les funcions que tracten amb l'entrada i sortida del sistema, des de llegir imatges fins a pintar els mapes de cobertura.
- **CUDA** (Figura 9.7): conté crides a l'API de CUDA englobades en macros, i templates, per capturar possibles errors i evitar haver d'escriure codi *boilerplate*.

i els *structs*:

- **Individual** (Figura 9.8): representa un individu de la població de l'algoritme genètic. Guarda els gens, en format vector, i el fitness, en format *float*.
- **AlgorithmParameters** (Figura 9.9): conté els paràmetres d'execució de l'algoritme de càlcul de mapes de cobertura.
- **OptimizationParameters** (Figura 9.10): conté els paràmetres d'execució de l'algoritme genètic.
- **SystemParameters** (Figura 9.11): conté els paràmetres d'entrada relacionats amb el funcionament del sistema que l'usuari ha entrat per la interfície.

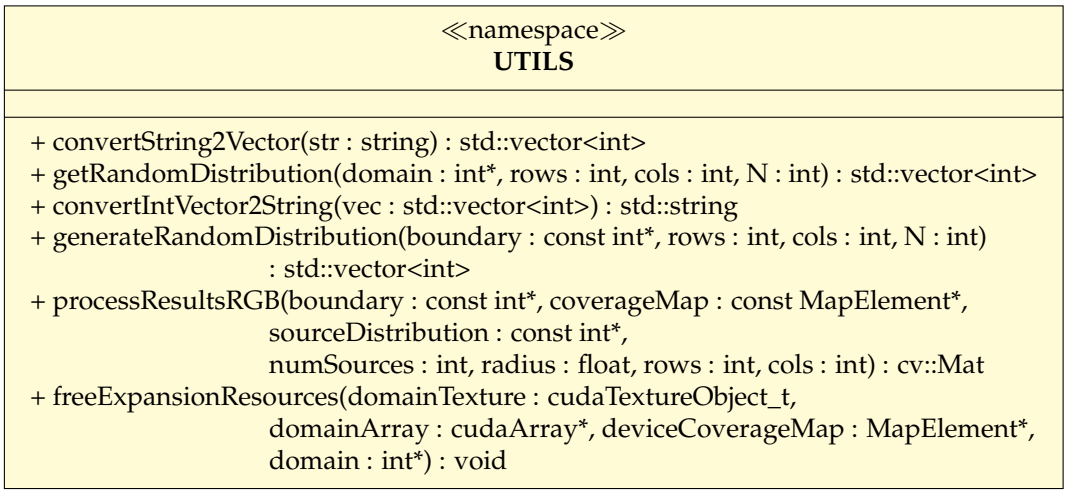


FIGURA 9.5: Diagrama namespace UTILS

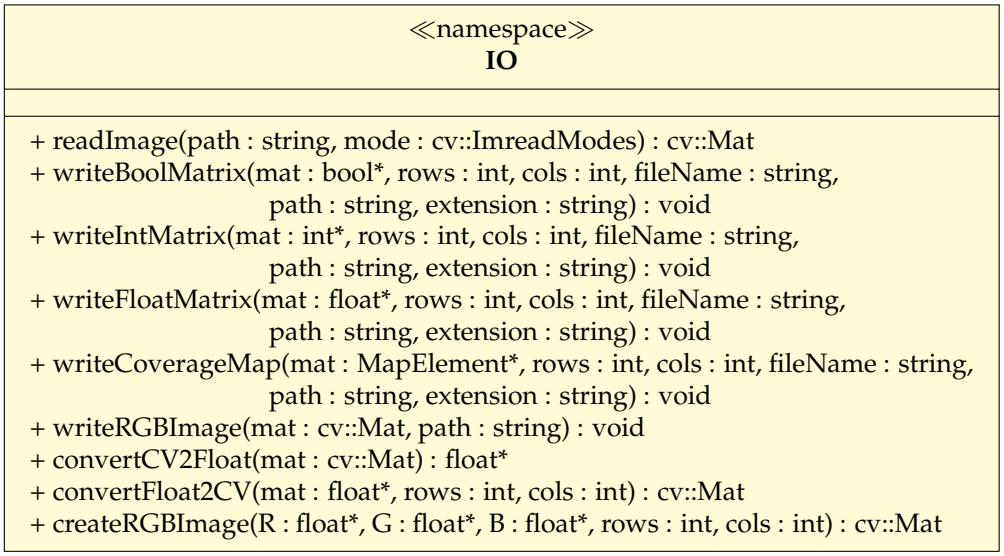


FIGURA 9.6: Diagrama namespace IO

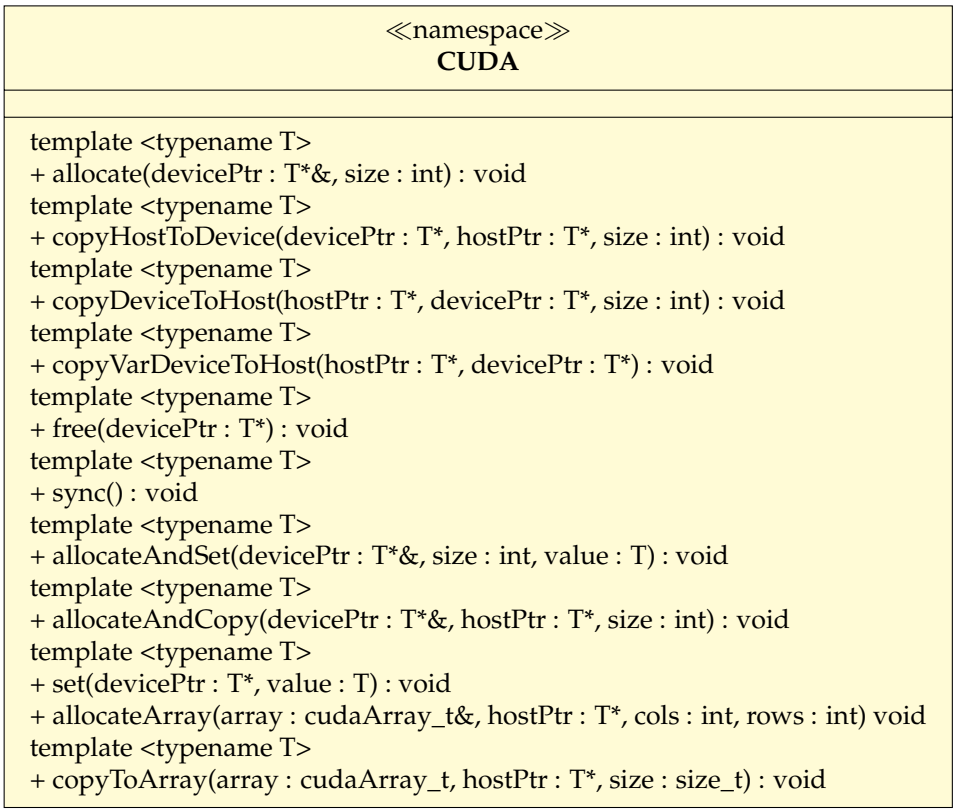


FIGURA 9.7: Diagrama namespace CUDA

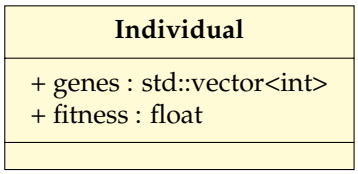


FIGURA 9.8: Diagrama de `Individual`

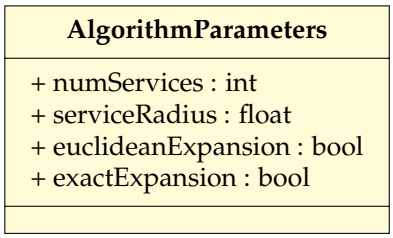


FIGURA 9.9: Diagrama de `AlgorithmParameters`

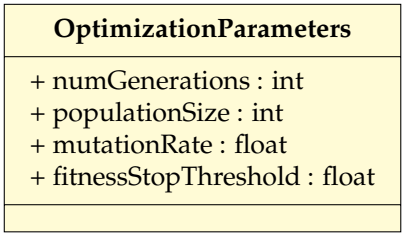


FIGURA 9.10: Diagrama de `OptimizationParameters`

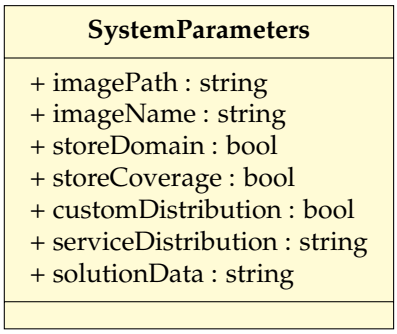


FIGURA 9.11: Diagrama de `SystemParameters`

Dins del fitxer `Coverage.h` tenim el *namespace* `COVERAGE` (Figura 9.12). Aquest conté tot el necessari per poder calcular mapes de cobertura, des de l'obtenció i conversió a textura del domini a partir d'una imatge fins al càlcul dels mapes i l'obtenció del percentatge de cobertura.

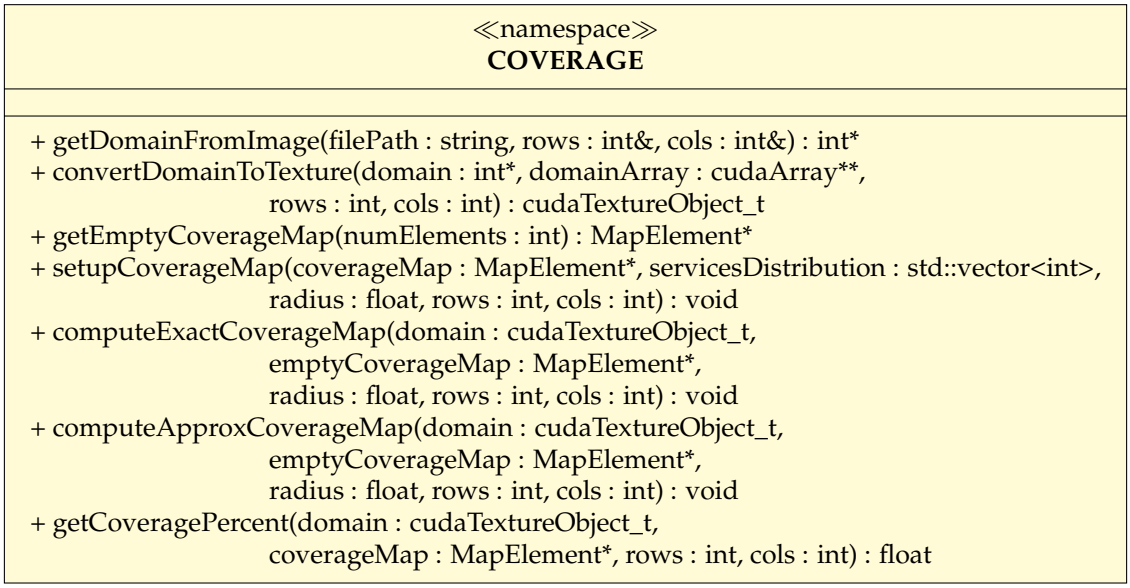


FIGURA 9.12: Diagrama namespace COVERAGE

Finalment, tenim la classe `MCLPSolver` (Figura 9.13) que representa l'algoritme genètic. Aquest només té una funció pública, `solve`, que inicia el procés d'optimització a partir dels paràmetres que se li hagin proporcionat en crear l'objecte.

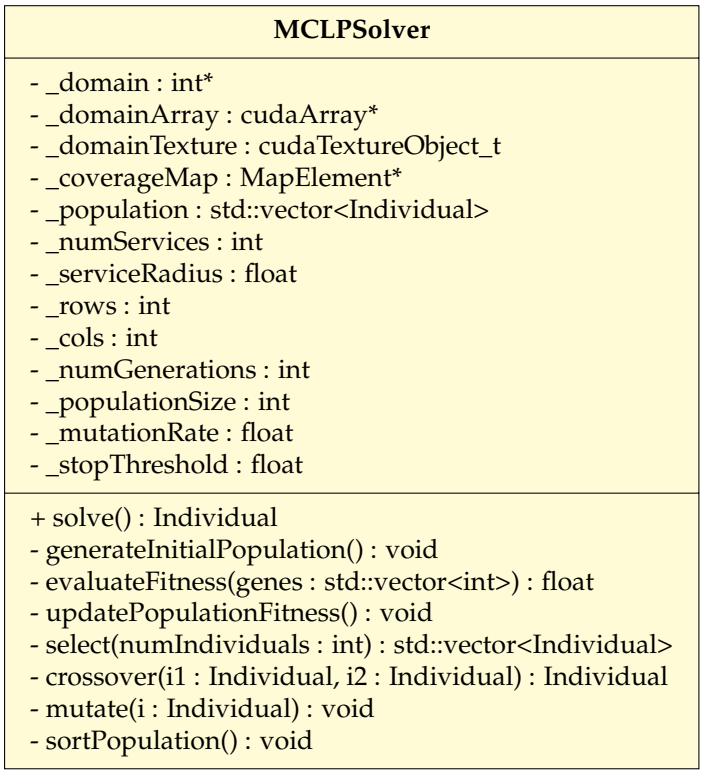


FIGURA 9.13: Diagrama classe `MCLPSolver`

Capítol 10

Implementació i proves

En aquesta secció s'exposen les implementacions principals de la solució proposada. El codi complet del projecte es troba disponible al següent repositori públic de GitHub.

10.1 Obtenció de dominis

El procés d'obtenció dels dominis s'inicia mitjançant la lectura de la imatge que els representa utilitzant la llibreria `OpenCV`. Aquesta es llegeix en format de matriu CV de tipus *32 bits single channel* (CV_32FC1), és a dir en blanc i negre. Aquesta elecció de format es fa per simplificar la detecció d'obstacles durant la descomposició en cel·les.

La descomposició en cel·les s'implementa mitjançant el *kernel* del Codi 10.1. En aquest, cada *thread* s'encarrega de processar un dels píxels de la imatge fent servir el següent criteri: si la intensitat del píxel analitzat és inferior o igual a 50.0 es marca la cel·la com a ocupada, en canvi, si aquesta és estrictament superior a 50.0, es comprova si la cel·la correspon a una cantonada (criteri exemplificat a la Figura 10.1) i es marca com a tal si ho és o en cas contrari com a lliure.

```
__global__ __inline__ void imageCellDecomposition(const float*
    grayscaleImage, int* domain, int rows, int cols) {
    int threadId = getThreadId();
    int numElements = rows * cols;

    if (threadId < numElements) {
        // Get the intensity of the pixel.
        float pixelIntensity = grayscaleImage[threadId];
        if (pixelIntensity <= 50.0f)
            // Assign -1 to the domain to represent a wall or obstacle.
            domain[threadId] = -1;
        else if (checkIfCorner(grayscaleImage, threadId, rows, cols))
            // Assign 1 to the domain to represent a free corner.
            domain[threadId] = 1;
        else
            // Assign 0 to the domain to represent a free point.
            domain[threadId] = 0;
    }
}
```

CODI 10.1: Kernel de preprocessat de dominis

Per determinar si un píxel correspon a una cantonada es comprova si té: més d'un veí marcat com a obstacle, però només un d'ells estigui en diagonal respecte al píxel analitzat. Si es tragués el primer criteri llavors, es marcarien vèrtexs d'obstacles com a cantonades, incorrectament, ja que un vèrtex no delimitarà mai un segment, almenys de manera precisa. A la Figura 10.1 es pot veure quins píxels queden marcats com a cantonada en un mur vertical.

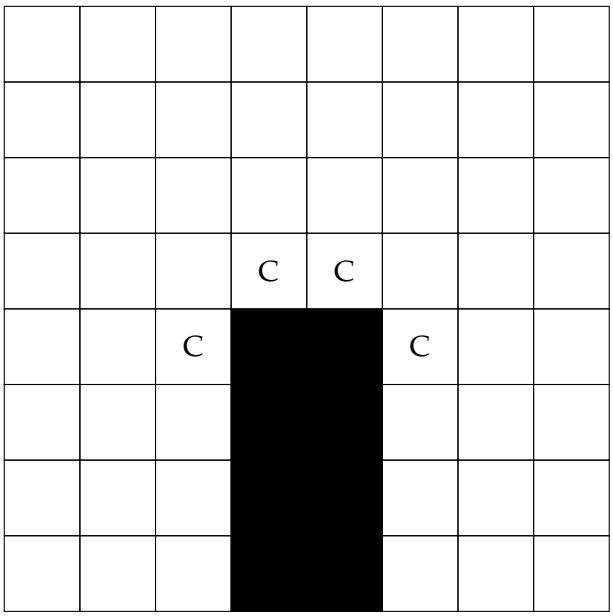


FIGURA 10.1: Exemplificació de quins píxels són marcats com a cantonada en una paret vertical.

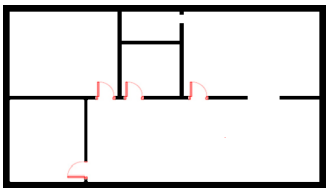
Un cop finalitzada la descomposició en cel·les només queda convertir el domini obtingut en format textura per accelerar les consultes que es fan a etapes posteriors. Aquesta conversió es pot observar al Codi 10.2.

```
cudaTextureObject_t COVERAGE::convertDomainToTexture(int* domain,
    cudaArray** domainArray, int rows, int cols){
    // Create channel format descriptor for integer type
    cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<int>();

    // Allocate memory for the device array
    CUDA::allocateArray(*domainArray, channelDesc, cols, rows);

    // Copy host domain to device array
    CUDA::copyToArray(*domainArray, domain, rows * cols);

    return CUDA::createTextureObject(*domainArray);
}

```

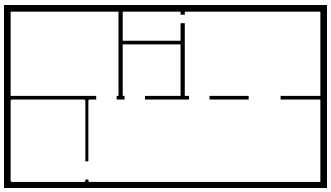
CODI 10.2: Funció que gestiona la conversió d'un domini en format `int*` a textura

10.1.1 Proves

Per tal de verificar l'adequat funcionament del procés d'obtenció de dominis, s'ha realitzat una sèrie de proves utilitzant cinc dominis diferents. Aquest conjunt de dominis està compost per dos entorns poligonals i tres plànols, cadascun d'ells amb dimensions i geometries úniques. A les figures 10.2 i 10.3 es poden observar els abans i després.

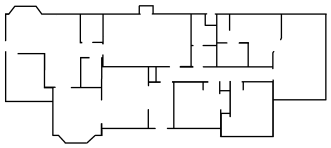


(A) Plànol 2 pre-descomposició en cel·les

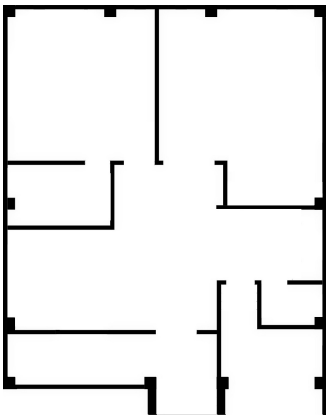


(B) Plànol 2 post-descomposició en cel·les

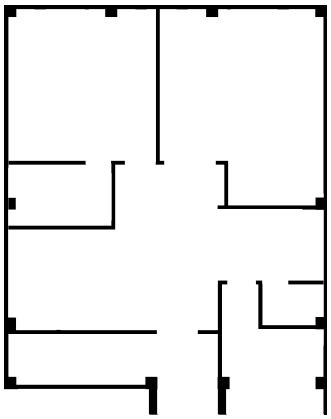
(C) Plànol 3 pre-descomposició en cel·les



(D) Plànol 3 post-descomposició en cel·les

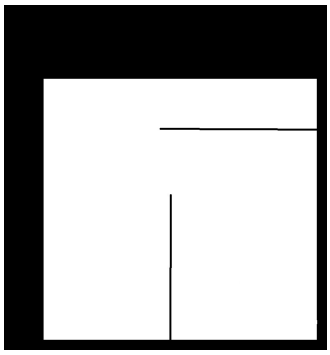


(E) Plànol 1 pre-descomposició en cel·les

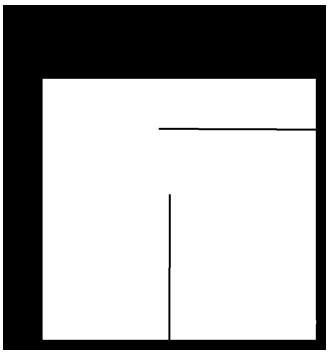


(F) Plànol 1 post-descomposició en cel·les

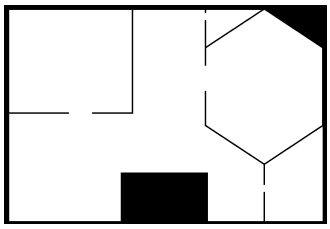
FIGURA 10.2: Obtenció dels plànols



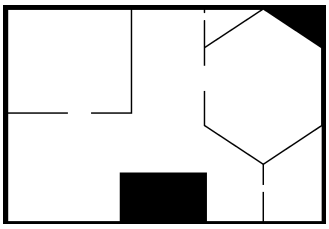
(A) Domini 1 pre-descomposició en cel·les



(B) Domini 1 post-descomposició en cel·les



(C) Domini 2 pre-descomposició en cel·les



(D) Domini 2 post-descomposició en cel·les

FIGURA 10.3: Obtenció dels dominis poligonals

10.2 Algoritme d'expansió pseudoeuclidiana

Per poder utilitzar l'algoritme de Bellman-Ford en paral·lel, implementat al (Codi 10.6), s'empra una combinació de sincronització a escala de bloc i global per assegurar un funcionament coherent.

En primer lloc, cada bloc fa el seu treball de manera independent, amb els seus *threads* executant-se en paral·lel i propagant les distàncies fins que no es puguin realitzar més canvis. No obstant això, per garantir la propagació entre blocs, ja que no tots els blocs s'executen simultàniament, s'efectua una sincronització global.

Un cop tots els blocs han finalitzat el seu treball, es verifica si almenys un d'ells ha dut a terme canvis en el mapa de cobertura. En cas afirmatiu, es crida de nou el *kernel* de propagació (Codi 10.3) per assegurar que tots els blocs estiguin sincronitzats i es pugui obtenir una solució completa.

Per a la sincronització a escala de bloc, s'empra una variable booleana emmagatzemada en memòria compartida. El *thread* $(0,0)$ del bloc la inicialitza a fals en cada iteració i és posada a cert pels *threads* que puguin propagar algun camí. També són necessaris dos punts de sincronització, un a l'inici del bucle (després d'establir la variable booleana a fals) i un a la seva fi, per garantir que tots els *threads* hagin finalitzat el procés de propagació abans de comprovar si la variable booleana s'ha posat a cert.

Per a la sincronització a escala global, s'utilitza una variable global que és inicialitzada a fals per la CPU en cada iteració i que és posada a cert pel *thread* $(0,0)$ dels blocs en els quals s'ha produït alguna propagació.

La matriu amb la qual es representa el mapa de cobertura està linealitzada en format *array*. Així un punter és suficient per accedir a tots els seus elements, que és més eficient que utilitzar-ne dos. Només s'ha de tenir en compte que a l'hora d'accedir als elements de la matriu és possible que s'hagin de fer conversions de coordenades bidimensionals a unidimensionals per accedir a certes posicions. Per exemple, per consultar cel·les contigües s'ha de convertir l'índex en qüestió a coordenades, sumar o restar 1 a les x i/o y i llavors tornar a convertir les coordenades a índex per accedir a la matriu.

10.2.1 Esquema de propagació

A continuació es mostra el *kernel* corresponent a la propagació de l'algoritme de Bellman-Ford. En aquest es pot observar la sincronització a escala de bloc amb la variable `blockChanges` que controla quan s'acaba l'execució d'aquest. També es pot apreciar com s'actualitza la variable que controla la sincronització global, `globalChanges`, que és actualitzada pel responsable en cas que hi hagi hagut algun canvi en el bloc.

Cada *thread* rep informació dels seus veïns mitjançant la finestra de consulta, que s'aplica una vegada a cada iteració garantint que en cas que hi hagi algun possible canvi potencial es valori.

```
__global__ void pseudoEuclideanExpansion(cudaTextureObject_t domainTex,
    MapElement* coverageMap, bool* globalChanges, int rows, int cols) {
    int tidX, tidY;
    get2DThreadId(tidX, tidY);

    int index = coordsToIndex(tidX, tidY, cols);
    bool responsible = threadIdx.x == 0 && threadIdx.y == 0;
    __shared__ bool blockChanges;

    do {
        if (responsible)
            // Set the value of blockChanges to false at the start of
    each iteration.
            blockChanges = false;

        __syncthreads();

        int tid = coordsToIndex(tidX, tidY, cols);
        int cellValue = tex2D<int>(domainTex, tidX, tidY);
        if (tidX < cols && tidY < rows && cellValue > -1) {
            bool updated = scanWindow(domainTex, coverageMap, tid, rows,
    cols);

            if (updated)
                blockChanges = true;
        }

        __syncthreads();
        if (blockChanges && responsible)
            *globalChanges = true;

    } while (blockChanges);
}
```

CODI 10.3: Kernel de propagació

Finestra de consulta i actualització de distàncies

Per consultar informació dels veïns, cada *thread* escaneja la finestra de consulta amb la funció `scanWindow` (Codi 10.4). Aquesta itera, en el sentit de les agulles del rellotge, per les vuit cel·les contigües i per cada una, en cas que no estigui ocupada per un obstacle, comprova si té informació útil mitjançant la crida `checkNeighInfo`, implementada al Codi 10.5.

En cas que la funció `checkNeighInfo` determini que efectivament com a mínim un dels veïns té informació útil, i l'utilitzi per actualitzar la distància guardada, llavors es marca com a cert la variable `updated` que al final de l'execució es retorna. Aquesta és llavors feta servir per determinar si hi ha hagut algun canvi a escala de bloc.

```
__device__ bool scanWindow(cudaTextureObject_t domainTex, MapElement*
    coverageMap, const int pixelIndex, int rows, int cols) {
    int ix = 0, iy = 0;
    indexToCoords(pixelIndex, ix, iy, cols);

    bool updated = false;

    // Loop through all the neighboring pixels.
    for (int i = -1; i < 2; i++) {
        for (int j = -1; j < 2; j++) {
            int newX = ix + i;
            int newY = iy + j;

            bool xInBounds = newX < cols && newX >= 0;
            bool yInBounds = newY < rows && newY >= 0;
            bool inBounds = xInBounds && yInBounds;

            if (inBounds) {
                int neighIndex = coordsToIndex(newX, newY, cols);

                // Check if it's a wall.
                int cellValue = tex2D<int>(domainTex, newX, newY);
                bool isWall = cellValue == -1;

                if (!isWall && pixelIndex != neighIndex) {
                    // Determine whether the neighboring pixel is
    diagonal.
                    bool diag = abs(i) == abs(j);

                    updated = updated || checkNeighInfo(distances,
    pixelIndex, neighIndex, diag);
                }
            }
        }
    }
    return updated;
}
```

CODI 10.4: Funció de consulta d'informació

Per determinar si el veí d'una cel·la proporciona millor informació de la qual té guardada s'aplica la funció `checkNeighInfo`. Aquesta comprova si la distància obtinguda a través veí és menor a la que té guardada, i en cas afirmatiu l'actualitza i retorna `true`, indicant que efectivament hi ha hagut un canvi.

```
__device__ bool checkNeighInfo(MapElement* coverageMap, int pointIndex,
    int neighIndex, bool diag) {
    // Compute the distance between the current point and the neighboring
    point.
    float distanceBetween = diag ? diagStepDistance() :
    unitStepDistance();
    bool updated = false;

    float currentDistance = coverageMap[pointIndex].distance;
    float tentativeDistance = coverageMap[neighIndex].distance +
    distanceBetween;

    if (tentativeDistance < currentDistance) {
        coverageMap[pointIndex].distance = tentativeDistance;

        updated = true;
    }

    return updated;
}
```

CODI 10.5: Funció de consulta de veïns

10.2.2 Bellman-Ford en paral·lel

Finalment, aplicant l'esquema de propagació descrit a les seccions anteriors, es pot implementar l'algoritme de Bellman-Ford tal com es mostra al Codi 10.6. En aquest es pot veure com s'inicialitza la variable de sincronització global `deviceFlag` i es va transferint entre CPU i GPU per controlar el procés. A més, es poden observar els paràmetres d'execució del *kernel*, que utilitza blocs de mida 16×16 i en crea tants com sigui necessari per cobrir tot el domini.

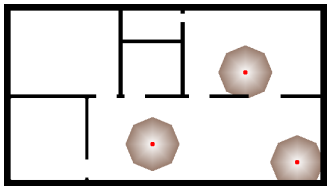
```
void parallelBellmanFord1(cudaTextureObject_t domainTexture, MapElement*
    deviceCoverageMap, int rows, int cols, float radius){
    bool hostFlag = false;
    bool* deviceFlag;
    int numElements = rows * cols;

    dim3 threadsPerBlock(16, 16);
    dim3 blocksPerGrid((cols + threadsPerBlock.x - 1) /
    threadsPerBlock.x, (rows + threadsPerBlock.y - 1) / threadsPerBlock.y);

    CUDA::allocate(deviceFlag, 1);
    do {
            CUDA::set(deviceFlag, false);
            pseudoEuclideanExpansion << <blocksPerGrid, threadsPerBlock
    >> > (domainTexture, deviceCoverageMap, deviceFlag, rows, cols);
            CUDA::sync();
            CUDA::copyDeviceToHost(&hostFlag, deviceFlag, 1);
    } while (hostFlag);
}
```

CODI 10.6: Implementació paral·lela de l'algoritme de Bellman-Ford

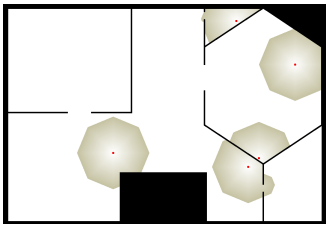
10.2.3 Proves

Durant les proves d'aquesta secció, es va detectar un error relacionat amb l'organització dels *threads*. En utilitzar una matriu linealitzada, es va assumir erròniament que la funció `getThreadId` també havia de retornar un índex linealitzat. Això és incorrecte, ja que si es calcula l'índex d'aquesta manera els blocs també queden linealitzats. Aquest error feia que la propagació fos extremadament ineficient, perquè a cada iteració només es propagava una línia. Per solucionar aquest problema es va crear una nova versió la funció `getThreadId`, `get2DThreadId` per organitzar els *threads* en blocs bidimensionals. A l'enllaç es pot veure una animació de com funcionava la propagació.

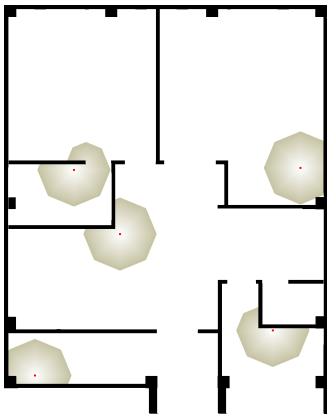
A continuació es mostren diversos mapes de cobertura obtinguts mitjançant l'algoritme descrit en aquesta secció. Com es pot observar, els resultats no són completament precisos, ja que els mapes haurien de tenir una forma circular en lloc d'octogonal.



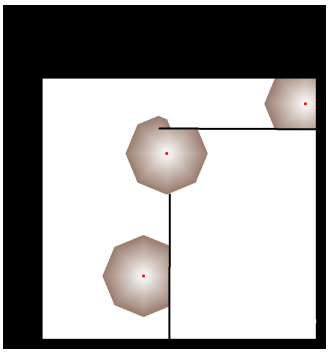
(A) Cobertura de distribució aleatòria al plànol 2



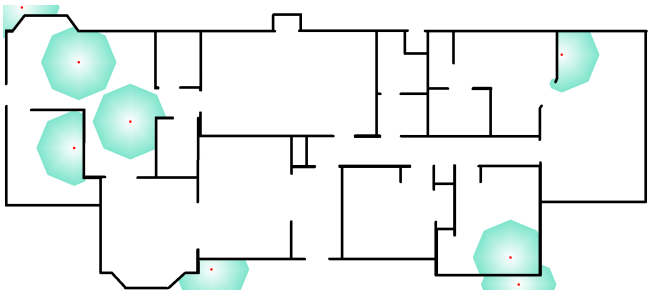
(B) Cobertura de distribució aleatòria al domini 2



(C) Cobertura de distribució aleatòria al plànol 1



(D) Cobertura de distribució aleatòria al domini 1



(E) Cobertura de distribució aleatòria al plànol 3

FIGURA 10.4: Cobertures de diferents distribucions de serveis

10.3 Algoritme d'expansió no restringida al graf

En aquesta secció s'explora la implementació de les modificacions de l'algoritme, descrites al Capítol 8, que milloren la precisió dels resultats obtinguts.

10.3.1 Esquema de propagació

L'esquema de propagació continua essent essencialment el mateix, però per completesa es mostra el codi igualment.

```
__global__ void euclideanExpansion(cudaTextureObject_t domainTex,
    MapElement* coverageMap, bool* globalChanges, int rows, int cols,
    float radius) {
    int tidX, tidY;
    get2DThreadId(tidX, tidY);

    int index = coordsToIndex(tidX, tidY, cols);
    bool responsible = threadIdx.x == 0 && threadIdx.y == 0;

    __shared__ bool blockChanges;

    do {
        if (responsible)
            // Set the value of blockIsActive to false at the start of
    each iteration
            blockChanges = false;

        __syncthreads();

        int tid = coordsToIndex(tidX, tidY, cols);

        int cellValue = tex2D<int>(domainTex, tidX, tidY);
        if (tidX < cols && tidY < rows && cellValue > -1) {
            bool updated =  listenUpdates(domainTex, coverageMap, tidX,
    tidY, rows, cols, radius);

            if (updated)
                blockChanges = true;
        }

        __syncthreads();

        if (blockChanges && responsible)
            *globalChanges = true;

    } while (blockChanges);
}
```

Finestra de consulta i actualització de distàncies i camins

En consultar la informació dels veïns mitjançant la finestra de consulta ja es poden observar els primers canvis respecte a la versió anterior. Aquí a la funció `checkNeighInfo` també s'hi passa informació relativa al predecessor del veí, que abans no s'utilitzava.

```
__device__ bool scanWindow(cudaTextureObject_t domainTex, MapElement*
    coverageMap, int pointIndex, int rows, int cols, float radius) {
    int ix = 0, iy = 0;
    indexToCoords(pointIndex, ix, iy, cols);

    bool updates = false;

    MapElement pointInfo = coverageMap[pointIndex];

    for (int dx = -1; dx < 2; dx++) {
        for (int dy = -1; dy < 2; dy++) {
            int nx = ix + dx;
            int ny = iy + dy;

            bool inBounds = nx < cols && ny < rows && nx >= 0 && ny >= 0;

            int neighIndex = coordsToIndex(nx, ny, cols);

            if (inBounds && neighIndex != pointIndex &&
    coverageMap[neighIndex].predecessor != -1) {
                int cellValue = tex2D<int>(domainTex, nx, ny);
                if (cellValue > -1) {
                    updates = updates || checkNeighInfo(domainTex,
    pointInfo, coverageMap[neighIndex], pointIndex, neighIndex, rows,
    cols, radius);
                }
            }
        }
    }

    if (updates)
        coverageMap[pointIndex] = pointInfo;

    return updates;
}
```

CODI 10.7: Funció de consulta d'informació

Aquí, per determinar si el veí proporciona millor informació a la qual es té guardada, s'utilitza el mateix mecanisme de comparació de distàncies (Codi 10.8). No obstant això, a diferència de la versió anterior, s'afegeix el càlcul i l'actualització del predecessor mitjançant la crida a la funció `suitablePredecessor` (Codi 10.9).

```
__device__ bool checkNeighInfo(cudaTextureObject_t domainTex,
    MapElement& pointInfo, MapElement neighInfo, int pointIndex, int
    neighIndex, int rows, int cols, float radius) {
    bool expanded = false;

    float currentDistance = pointInfo.distance;
    float tentativeDistance = neighInfo.distance +
    indexDistance(pointIndex, neighIndex, cols);

    if (tentativeDistance < currentDistance) {
        int predecessor = suitablePredecessor(domainTex, pointIndex,
    neighIndex, neighInfo.predecessor, rows, cols);

        MapElement newInfo{ tentativeDistance, predecessor,
    neighInfo.source };

        pointInfo = newInfo;

        expanded = true;
    }

    return expanded;
}
```

CODI 10.8: Funció de consulta de veïns

Per seleccionar el predecessor (Codi 10.9), es comprova si el veí que proporciona la informació és una cantonada i es verifica si el seu predecessor és visible des del punt en qüestió, mitjançant el test de visibilitat (Codi 10.10). Si una és cantonada o el seu predecessor no és visible, retorna el veí com a predecessor. En cas contrari retorna el predecessor del veí.

```
__device__ int suitablePredecessor(cudaTextureObject_t domainTex, int
    pointIndex, int neighIndex, int neighPredecessorIndex, int rows, int
    cols) {
    int neighX, neighY;
    indexToCoords(neighIndex, neighX, neighY, cols);

    int predX, predY;
    indexToCoords(neighPredecessorIndex, predX, predY, cols);

    int currentX, currentY;
    indexToCoords(pointIndex, currentX, currentY, cols);

    bool neighIsCorner = tex2D<int>(domainTex, neighX, neighY);
    bool neighPredIsVisible = visibilityTest(domainTex, rows, cols,
    predX, predY, currentX, currentY);

    if (neighIsCorner || !neighPredIsVisible)
        return neighIndex;
    else
        return neighPredecessorIndex;
}
```

CODI 10.9: Funció de tria de predecessor

```
__device__  bool visibilityTest(cudaTextureObject_t domainTex, int rows,
    int cols, int oX, int oY, int gX, int gY) {
    int dx = abs(gX - oX);
    int dy = abs(gY - oY);
    int sx = (oX < gX) ? 1 : -1;
    int sy = (oY < gY) ? 1 : -1;
    int err = dx - dy;

    while (true) {
        int cellValue = tex2D<int>(domainTex, oX, oY);
        if (cellValue == -1)
            return false;

        if (oX == gX && oY == gY)
            break;

        int e2 = 2 * err;

        if (e2 > -dy) {
            err -= dy;
            oX += sx;
        }

        if (e2 < dx) {
            err += dx;
            oY += sy;
        }

        if (oX >= cols || oX < 0 || oY >= rows || oY < 0)
            return false;
    }

    return true;
}
```

CODI 10.10: Funció que computa el test de visibilitat

La correcció de distàncies es fa a partir del *kernel* descrit al Codi 10.11. Aquest calcula la **EEDT** de tots els punts coberts cap al servei que els cobreix mitjançant la crida `computeDistance` (Codi 10.12), que reconstrueix el camí que porta a la seu a partir dels segments delimitats pels predecessors i retorna la seva longitud.

```
__global__ void EEDT(MapElement* coverageMap, bool* globalChanges, int
    rows, int cols, float radius) {
    int tidX, tidY;
    get2DThreadId(tidX, tidY);

    if (tidX >= cols || tidY >= rows)
        return;

    int tid = coordsToIndex(tidX, tidY, cols);
    MapElement pointInfo = coverageMap[tid];

    if (pointInfo.distance <= radius) {
        float exactDistance = computeDistance(coverageMap, tid, cols);

        if (exactDistance != -1 && exactDistance < pointInfo.distance) {
            coverageMap[tid].distance = exactDistance;

            *globalChanges = true;
        }
    }
}
```

CODI 10.11: Kernel que actualitza les distàncies del mapa de cobertura amb les exactes

Per obtenir la distància, la funció `computeDistance` va sumant les longituds dels segments que formen el camí, tal com es descriu a l'Equació 8.2 del Capítol 8, a partir de la funció `indexDistance` que calcula la distància euclidiana entre dos índexs linealitzats d'una matriu (Codi 10.13).

```
__device__ __inline__ float computeDistance(MapElement* distanceMap, int
    pointIndex, int cols) {
    float distance = 0.0f;

    int currentPoint = pointIndex;
    int nextGoal = distanceMap[currentPoint].predecessor;

    while (nextGoal != currentPoint) {
        distance += indexDistance(currentPoint, nextGoal, cols);
        currentPoint = nextGoal;
        nextGoal = distanceMap[currentPoint].predecessor;
    }

    return distance;
}
```

CODI 10.12: Funció que calcula la longitud dels camins

```
__device__ __inline__ float indexDistance(int pointA, int pointB, int
    cols) {
    int xA, xB, yA, yB;
    indexToCoords(pointA, xA, yA, cols);
    indexToCoords(pointB, xB, yB, cols);

    return sqrtf((xA - xB) * (xA - xB) + (yA - yB) * (yA - yB));
}
```

CODI 10.13: Funció que calcula la distància euclidiana entre dos índexs linealitzats d'una matriu

10.3.2 Bellman-Ford en paral·lel amb camins no restringits al graf

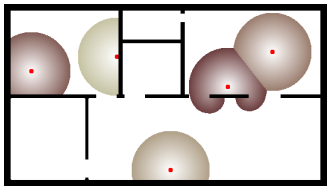
La introducció de la fase de correcció de distàncies es veu reflectida a l'algoritme de la següent manera (Codi 10.14). S'engloba la propagació anterior en un bucle que executa aquesta al final de cada propagació i el procés només acaba quan cap de les dues fases ha pogut fer un canvi.

```
1  void parallelBellmanFord2(cudaTextureObject_t domainTexture, MapElement*
       deviceCoverageMap, int rows, int cols, float radius) {
2      bool hostFlag = false;
3      bool* deviceFlag;
4
5      int numElements = rows * cols;
6
7      dim3 threadsPerBlock(16, 16);
8      dim3 blocksPerGrid((cols + threadsPerBlock.x - 1) /
       threadsPerBlock.x, (rows + threadsPerBlock.y - 1) / threadsPerBlock.y);
9
10     CUDA::allocate(deviceFlag, 1);
11     do {
12         do {
13             CUDA::set(deviceFlag, false);
14             euclideanExpansion << <blocksPerGrid, threadsPerBlock >> >
       (domainTexture, deviceCoverageMap, deviceFlag, rows, cols, radius);
15             CUDA::sync();
16             CUDA::copyDeviceToHost(&hostFlag, deviceFlag, 1);
17         } while (hostFlag);
18
19         CUDA::set(deviceFlag, false);
20         EEDT << <blocksPerGrid, threadsPerBlock >> > (deviceCoverageMap,
       deviceFlag, rows, cols, radius);
21         CUDA::sync();
22         CUDA::copyDeviceToHost(&hostFlag, deviceFlag, 1);
23     } while (hostFlag);
24
25     CUDA::free(deviceFlag);
26 }
```

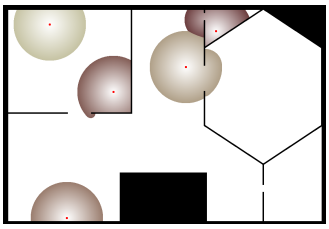
CODI 10.14: Implementació paral·lela de l'algortime de Bellman-Ford amb camins no restringits al graf

10.3.3 Proves

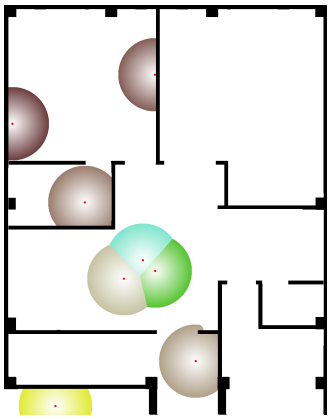
A continuació es mostren diversos mapes de cobertura obtinguts mitjançant l'adaptació de l'algoritme que s'ha descrit en aquesta secció. Com es pot observar, els resultats milloren considerablement i són molt més realistes.



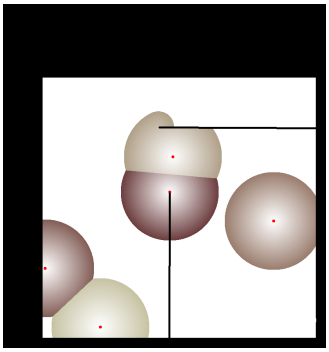
(A) Cobertura de distribució aleatòria al plànol 2



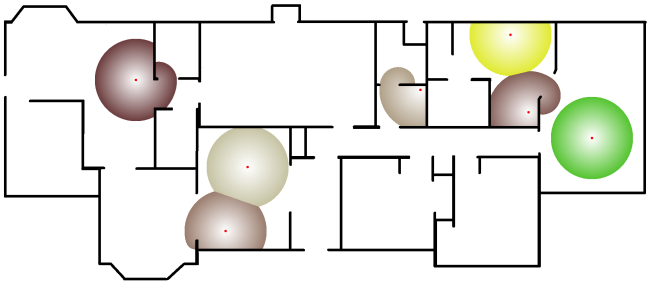
(B) Cobertura de distribució aleatòria al domini 2



(C) Cobertura de distribució aleatòria al plànol 1



(D) Cobertura de distribució aleatòria al domini 1



(E) Cobertura de distribució aleatòria al plànol 3

FIGURA 10.5: Cobertures de diferents distribucions de serveis

10.4 Algoritme genètic

En aquesta secció es presenta la implementació de les operacions principals de l'algoritme genètic, incloent-hi el procés d'evolució, que s'utilitza per trobar solucions òptimes del MCLP.

10.4.1 Selecció

L'operació de selecció implementada es basa en la repetició d'un torneig, conegut a la literatura com a *tournament selection*, tantes vegades com individus es vulguin seleccionar. A cada un es trien tres individus aleatoris de la població on el que té major fitness guanya i s'afegeix al grup de seleccionats.

```
std::vector<Individual> MCLPSolver::selec(int numIndividuals){
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(0, _populationSize - 1);

    std::vector<Individual> selectedIndividuals;
    selectedIndividuals.reserve(numIndividuals);

    for (int i = 0; i < numIndividuals; ++i) {
        // Perform tournament selection
        int tournamentSize = 3;
        std::vector<int> tournamentIndices(tournamentSize);

        // Select random individuals for the tournament
        for (int j = 0; j < tournamentSize; ++j) {
            tournamentIndices[j] = dis(gen);
        }

        // Find the individual with the highest fitness in the tournament
        Individual winner = _population[tournamentIndices[0]];
        for (int j = 1; j < tournamentSize; ++j) {
            Individual& contender = _population[tournamentIndices[j]];
            if (contender.fitness > winner.fitness) {
                winner = contender;
            }
        }

        // Add the winner to the selected individuals
        selectedIndividuals.push_back(winner);
    }

    return selectedIndividuals;
}
```

CODI 10.15: Funció de selecció per torneig

10.4.2 Reproducció

La reproducció es basa en l'aplicació del que es coneix a la literatura com a *one-point crossover*. Aquest procés consisteix a prendre dos individus de la població, seleccionar un punt d'intersecció i a partir d'aquest crear-ne un de nou com a combinació d'una de les particions de cada un. Obtenint així un individu que hereta propietats de tots dos.

En aquest cas particular, el punt d'intersecció determina quines posicions de serveis rep de cada `parent`. Amb el punt d'intersecció a la posició N, el nou individu tindrà: les N primeres coordenades igual que el *parent1*, i els restants igual que el `parent2`.

```
Individual MCLPSolver::crossover(const Individual& parent1, const
    Individual& parent2) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(0, parent1.genes.size() - 1);

    // Select a random crossover point
    int crossoverPoint = dis(gen);

    // Create a child individual
    Individual child;
    child.genes.reserve(parent1.genes.size());

    // Copy genes from parent1 up to the crossover point
    for (int i = 0; i < crossoverPoint; ++i) {
        child.genes.push_back(parent1.genes[i]);
    }

    // Copy genes from parent2 after the crossover point
    for (int i = crossoverPoint; i < parent1.genes.size(); ++i) {
        child.genes.push_back(parent2.genes[i]);
    }

    return child;
}
```

CODI 10.16: Funció de reproducció mitjançant *one-point crossover*

10.4.3 Mutació

Per la mutació d'un individu s'escull un índex aleatori i es modifica la coordenada corresponent a aquella posició entre un 1 i un 15 per cent, triat també de manera aleatòria.

Restringir la mutació a canvis relatius petits, com es fa en aquest cas, permet a l'algoritme explorar l'espai de cerca amb més precisió. En aquest enfocament, al mutar una solució, no se n'obté una completament diferent, sinó una de similar. Aquest enfocament és considerat superior a fer canvis més extrems, tal com es menciona a Koza 1992, ja que complementa la cerca realitzada mitjançant el *crossover*, que normalment ja és més que suficient i no requereix mutacions, proporcionant una major varietat de solucions dins d'un rang de *fitness* específic, en lloc de generar solucions radicalment diferents.

```
void MCLPSolver::mutate(Individual& individual) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> mutationIndexDist(0,
    individual.genes.size() - 1);
    std::uniform_real_distribution<> mutationValueDist(0.01, 0.15); //
    Varying mutation percentage between 1% and 15%

    int mutationIndex = mutationIndexDist(gen);
    int geneValue = individual.genes[mutationIndex];
    int mutationRange = static_cast<int>(std::round(geneValue *
    mutationValueDist(gen)));

    if (mutationIndex % 2 == 0) {
        // Mutate X value
        int newXValue = std::max(0, std::min(_cols, geneValue +
    mutationRange));
        individual.genes[mutationIndex] = newXValue;
    }
    else {
        // Mutate Y value
        int newYValue = std::max(0, std::min(_rows, geneValue +
    mutationRange));
        individual.genes[mutationIndex] = newYValue;
    }

    individual.fitness = 0.0f;
}
```

CODI 10.17: Funció de mutació

10.4.4 Càlcul del fitness

Abans de fer qualsevol de les operacions anteriors s'assumeix que cada individu de la població ja té calculat el seu corresponent fitness. Això s'obté següent funció, que itera per la població actualitzant el fitness dels individus que tenen marcat com a zero, que dins del bucle d'evolució indica que han patit una mutació o que han sigut novament introduïts a la població:

```
void MCLPSolver::updatePopulationFitness(){
    for (int i = 0; i < _populationSize; i++) {
        if (_population[i].fitness == 0)
            _population[i].fitness =
    evaluateFitness(_population[i].genes);
    }
}
```

CODI 10.18: Funció d'actualització del fitness de la població

Per calcular el fitness d'individus s'utilitza la següent funció, que calcula el percentatge de cobertura al mapa amb la distribució de serveis que especifiquen els seus gens:

```
float MCLPSolver::evaluateFitness(std::vector<int> genes) {
    // Setup the coverageMap
    COVERAGE::setupCoverageMap(_coverageMap, genes, _radius, _rows,
    _cols);

    COVERAGE::computeExactCoverageMap(_domainTexture, _coverageMap,
    _radius, _rows, _cols);

    return COVERAGE::getCoveragePercent(_domainTexture, _coverageMap,
    _rows, _cols);
}
```

CODI 10.19: Funció de fitness

10.4.5 Esquema d'evolució

La crida `solve` de la classe `MCLPSolver` representa el bucle d'evolució de l'algoritme genètic (Codi 10.20). En aquest s'utilitzen totes les operacions mencionades en aquesta secció per explorar l'espai de cerca de possibles solucions amb l'objectiu de trobar la millor possible.

El procés d'evolució utilitzat segueix la següent seqüència d'operacions:

1. Selecció d'individus reproductors.
2. Reproducció dels individus seleccionats.
3. Possible mutació dels fills creats.
4. Mutació d'un 10% d'individus de la població, triats a l'atzar.
5. Substitució dels individus amb menys fitness de la població pels fills de les parelles reproductores.
6. Càlcul del fitness dels nous individus.

Tot i seguir l'esquema general dels algoritmes genètics hi ha una fase addicional que s'ha introduït després d'experimentar amb diferents estratègies i valorar els resultats empírics. Aquesta és la número 4.

La mutació d'un percentatge aleatori d'individus de la població s'ha trobat que contribueix a fer que el millor individu de la població no s'encalli a solucions subòptimes. Abans d'introduir aquesta fase hi havia vegades on la millor solució quedava encallada a un cert percentatge de cobertura durant un alt nombre d'iteracions.

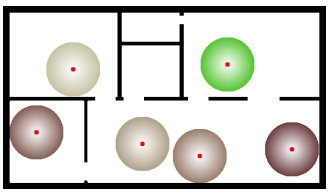
L'únic punt en contra que té aquesta fase és que introdueix un molt petit risc de perdre la millor solució a causa d'una mutació. Tot i que sigui una possibilitat real no s'ha trobat que afecti els resultats obtinguts de cap manera rellevant, ja que en cas de donar-se la situació, la mutació és prou petita per afectar poc o poder ser corregida més endavant.

```
Individual MCLPSolver::solve(){
    int currentGeneration = 0;
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dis(0.0, 1.0);

    Individual fittestIndividual = _population[0];

    while (currentGeneration < _numGenerations &&
    fittestIndividual.fitness < _stopThreshold) {
        int numParents = _populationSize % 2 == 0
                ? 1 + _populationSize / 2
                : _populationSize / 2;

        std::vector<Individual> parents = select(numParents);
        std::vector<Individual> children;

        children.reserve(parents.size() / 2);

        int j = 1;

        for (int i = 0; i < numParents; i+=2) {
            Individual parent1 = parents[i];
            Individual parent2 = parents[i + 1];
            // Create a child using one-point crossover
            Individual child = crossover(parent1, parent2);

            // Mutate the child based on mutation probability
            if (dis(gen) < _mutationRate) {
                mutate(child);
            }
            children.push_back(child);
            j++;
        }

        // Mutate 10% random individuals from the population
        int numRandomMutations = static_cast<int>
                            (std::round(0.1 * _populationSize));

        for (int i = 0; i < numRandomMutations; i++) {
            int randomIndex = std::uniform_int_distribution<>
                                (0, _populationSize - 1)(gen);
            mutate(_population[randomIndex]);
        }

        j = 0;
        for (int i = _populationSize - children.size();
                                i < _populationSize; i++) {
            _population[i] = children[j];
            j++;
        }

        updatePopulationFitness();
        sortPopulation();
        fittestIndividual = _population[0];
        ++currentGeneration;
    }

    return fittestIndividual;
}
```

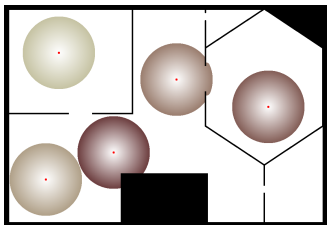
CODI 10.20: Funció d'evolució

10.4.6 Proves

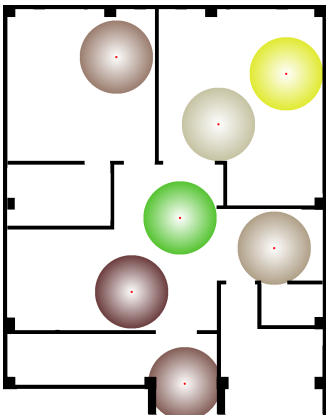
A continuació es mostren una sèrie de distribucions optimes, de diferent nombre de serveis, per cada un del dominis obtingudes a partir de l'algoritme genètic. Per cada una d'elles s'han utilitzat poblacions de 50 individus, 100 generacions i un 10% de probabilitat de mutació.



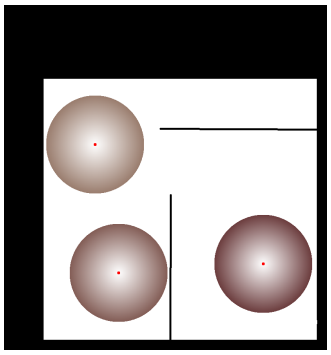
(A) Cobertura de distribució òptima al plànol 2



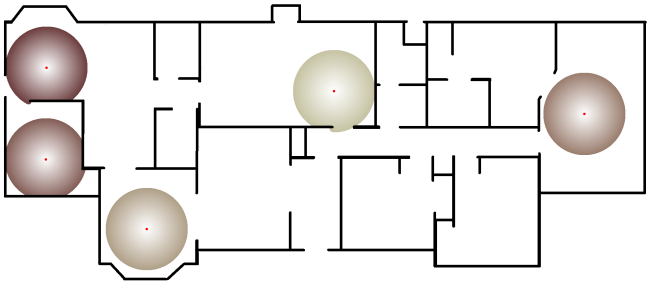
(B) Cobertura de distribució òptima al domini 2



(C) Cobertura de distribució òptima al plànol 1



(D) Cobertura de distribució òptima al domini 1



(E) Cobertura de distribució òptima al plànol 3

FIGURA 10.6: Cobertures de diferents distribucions òptimes de serveis

Capítol 11

Resultats

Per recol·lectar els resultats presentats en aquesta secció s'ha utilitzat un equip amb les següents característiques:

Component	Especificacions
CPU	Intel Core i5-6600
RAM	28 GB DDR4 3600MHz
GPU	NVIDIA GeForce GTX 1050

TAULA 11.1: Hardware utilitzat

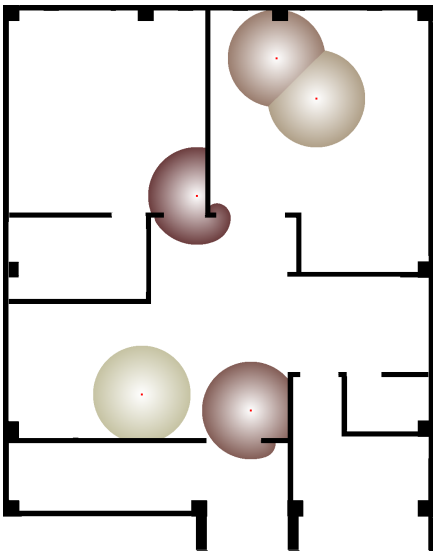
11.1 Càlcul de mapes de cobertura en entorns amb obstacles

A la Taula 11.2 es mostren les mètriques de rendiment obtingudes en calcular mapes de cobertura en diversos entorns amb obstacles. A la Figura 11.1 es poden observar els mapes de cobertura corresponents a cada una de les entrades de la taula anterior.

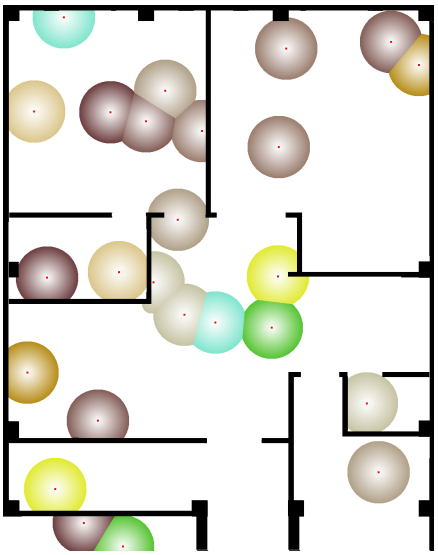
Domini	Serveis	Radi d'abast	Iteracions	Temps d'execució (s)
Plànol 1	5	125.0	6	0.234
Plànol 1	25	80.0	11	0.252
Plànol 3	5	250.0	66	0.941
Plànol 3	50	50.0	16	0.333
Laberint	1	1200.0	28	0.713
Laberint	40	175.0	41	0.572

TAULA 11.2: Rendiment càlcul de mapes de cobertura per diferents dominis i configuracions

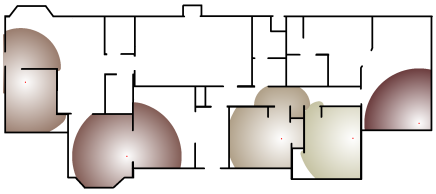
Els resultats d'aquest apartat es considera que són satisfactoris, ja que tots els mapes de cobertura són correctes i es calculen en un temps raonable. Tot i que es podrien calcular de manera més eficient arreglant el problema de propagacions il·legals i hi hagi alguns casos on l'algoritme fa més iteracions del compte sense que hi hagi canvis al mapa de cobertura, tal com es pot veure a l'animació que s'enllaça al final d'aquesta secció.



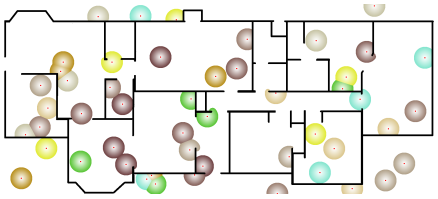
(A) Plànol 1, 5 serveis amb 125 de rang



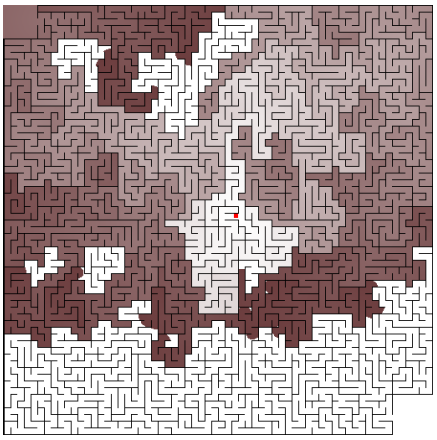
(B) Plànol 1, 25 serveis amb 80 de rang



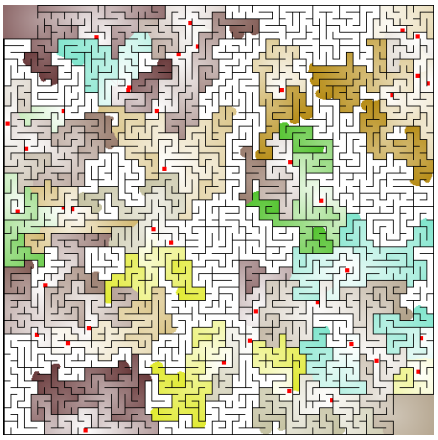
(C) Plànol 3, 5 serveis amb 250 de rang



(D) Plànol 3, 50 serveis amb 50 de rang



(E) Laberint, 1 servei amb 1200 de rang



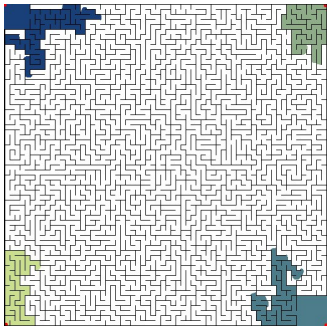
(F) Laberint, 40 serveis amb 175 de rang

FIGURA 11.1: Mapes de cobertura especificats a la taula de rendiment, Taula 11.2

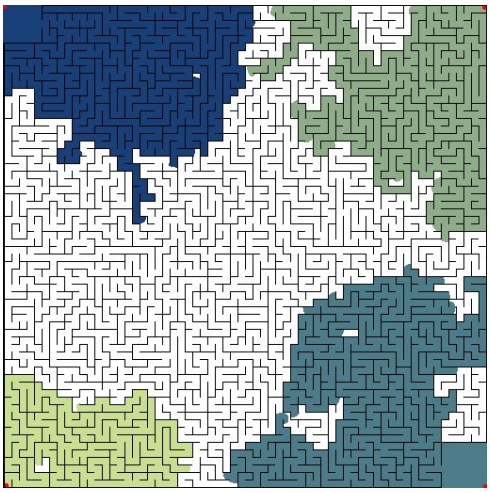
Per mostrar una de les possibles aplicacions d'aquest mètode es presenta un diagrama de Voronoi de quatre regions d'un laberint (Figura 11.2d) obtingut a partir d'un mapa de cobertura.

Un diagrama de Voronoi és un tipus de partició de dominis amb seus, on cada punt té associada aquella que li queda més propera. Per tant, queden tantes particions com seus hi hagi i cada una conté els punts que li queden més a prop que a la resta.

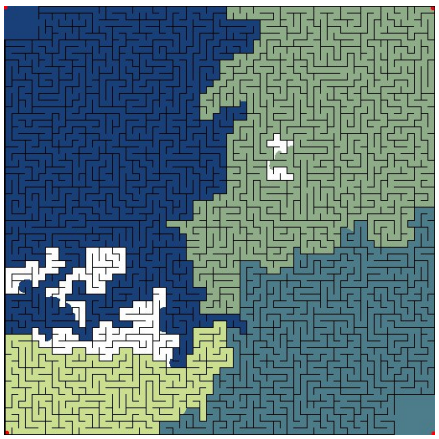
Per aconseguir-lo s'han col·locat quatre seus, una a cada cantonada, amb un radi prou alt percobrir tot el domini. A la Figura 11.2 es pot veure com evoluciona el mapa de cobertura fins a convergir en el diagrama de Voronoi.



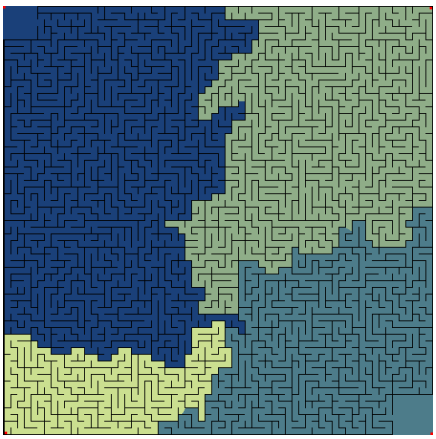
(A) Cobertura fins a 100 píxels d'abast



(B) Cobertura fins a 500 píxels d'abast



(C) Cobertura fins a 1000 píxels d'abast



(D) Cobertura fins a 1200 píxels d'abast

FIGURA 11.2: Evolució de mapa de cobertura de quatre seus, una a cada cantonada, amb 1200 de radi d'abast

També és possible utilitzar aquest mètode per explorar laberints, tal com es pot veure a l'animació de l'enllaç. En aquesta es mostra l'evolució de la cobertura d'un servei col·locat a la posició $(0, 0)$, amb un de rang 10^3 píxels, on cada *frame* correspon a una iteració de l'algoritme.

11.2 Cerca de solucions del MCLP

Per avaluar el rendiment de l'algoritme genètic dissenyat s'han buscat tres solucions òptimes diferents per cada un dels plànols mostrats anteriorment. A les taules que es mostren a continuació es poden veure les configuracions utilitzades i mètriques de rendiment.

En tots els casos s'ha utilitzat una probabilitat de mutació del 20% i una població de 50.

Per al plànol 1 (11.2.1) s'han realitzat les proves descrites a la Taula 11.3.

Serveis	Radi	Generacions	Cobertura (%)	Temps d'execució (s)
3	150	10	14.9213	37.12
5	125	30	17.2453	60.23
10	105	50	24.04	74.63

TAULA 11.3: Rendiment algoritme genètic pel plànol 1

Per al plànol 2 (11.2.2) s'han realitzat les proves descrites a la Taula 11.4.

Serveis	Radi	Generacions	Cobertura (%)	Temps d'execució (s)
5	50	20	28.0541	5.028
20	50	20	75.089	10.64
50	50	100	99.98	61.83

TAULA 11.4: Rendiment algoritme genètic pel plànol 2

Per al plànol 3 (11.2.3) s'han realitzat les proves descrites a la Taula 11.5.

Serveis	Radi	Generacions	Cobertura (%)	Temps d'execució (s)
5	125	20	15.0564	44.11
10	100	20	19.3085	51.32
20	100	20	32.7782	94.58

TAULA 11.5: Rendiment algoritme genètic pel plànol 3

Els resultats d'aquesta secció es considera que són satisfactoris, però millorables. Ja que els temps d'execució són relativament alts, a causa del mecanisme que evita propagacions il·legals, la qual cosa provoca que no es puguin utilitzar mides de poblacions i iteracions gaire elevades.

11.2.1 Plànol 1

A la Figura 11.3 es pot veure el mapa de cobertura obtingut amb la configuració a la primera fila de la Taula 11.3 de l'algoritme genètic. Tal com es pot apreciar, és un dels millors possibles, ja que totes les seus estan col·locades de tal manera que cap té la cobertura blocada per cap obstacle. A la Figura 11.4 es pot observar l'evolució del fitness al llarg de les iteracions.

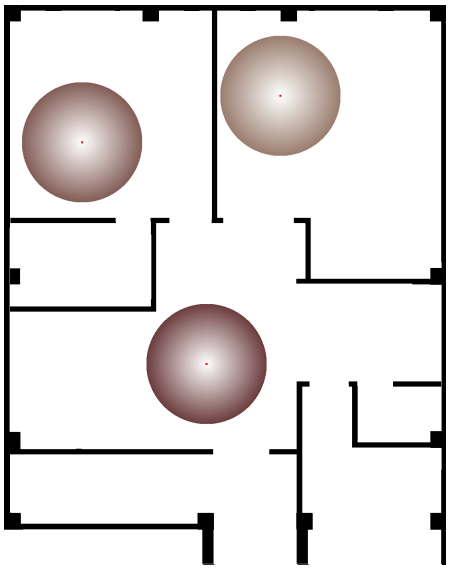


FIGURA 11.3: Mapa de cobertura òptim per 3 serveis amb 150 de radi

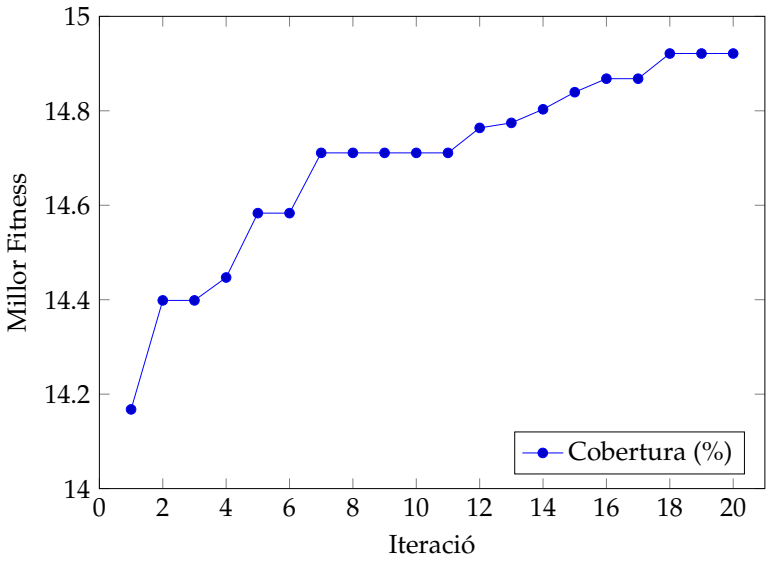


FIGURA 11.4: Evolució del fitness de la millor solució del plànol 1 amb la configuració de la primera fila de la Taula 11.3

A la Figura 11.5 es pot veure el mapa de cobertura obtingut amb la configuració a la segona fila de la Taula 11.3 de l'algoritme genètic. Igual que en el cas anterior aquesta és un dels millors possibles, ja que totes les seus estan col·locades de tal manera que cap té la cobertura blocada per cap obstacle. A la Figura 11.6 es pot observar l'evolució del fitness al llarg de les iteracions.

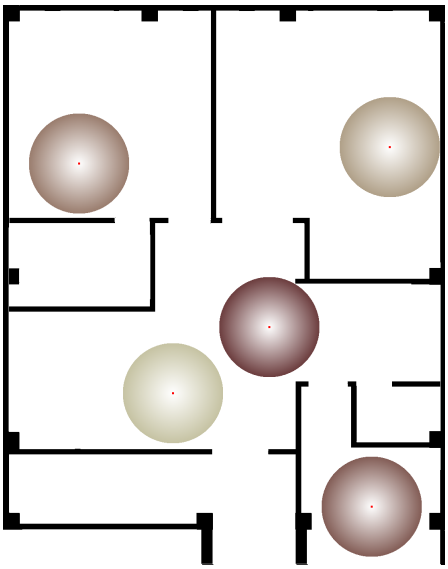


FIGURA 11.5: Mapa de cobertura òptim per 5 serveis amb 150 de radi

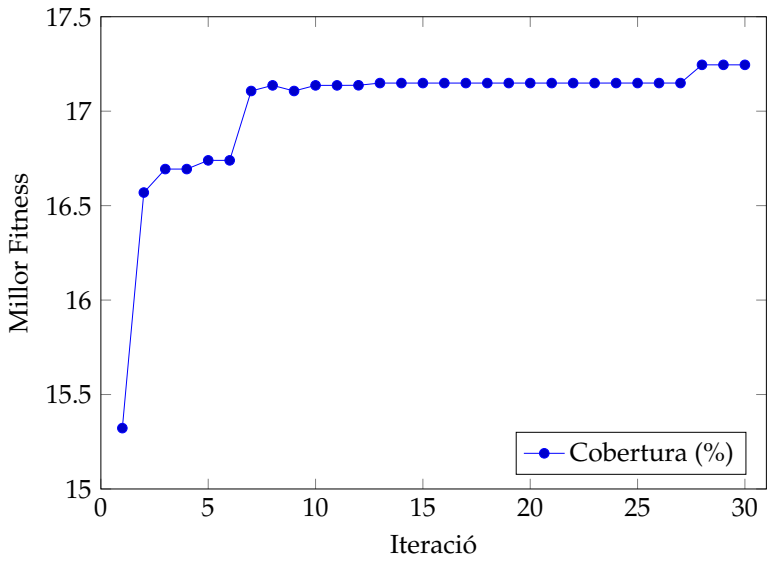


FIGURA 11.6: Evolució del fitness de la millor solució del plànol 1 amb la configuració de la segona fila de la Taula 11.3

A la Figura 11.7 es pot veure el mapa de cobertura obtingut amb la configuració a la tercera fila de la Taula 11.3 de l'algoritme genètic. Tal com es pot apreciar, no és de les millors possibles perquè hi ha serveis que podries estar a una millor posició, com el de dalt a l'esquerra i els tres de la zona inferior; tot i això, la solució és força bona. A la Figura 11.8 es pot observar l'evolució del fitness al llarg de les iteracions.

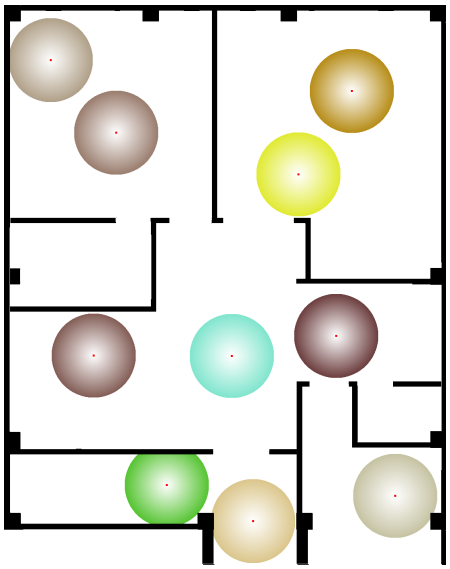


FIGURA 11.7: Mapa de cobertura quasi òptim per 10 serveis amb 105 de radi

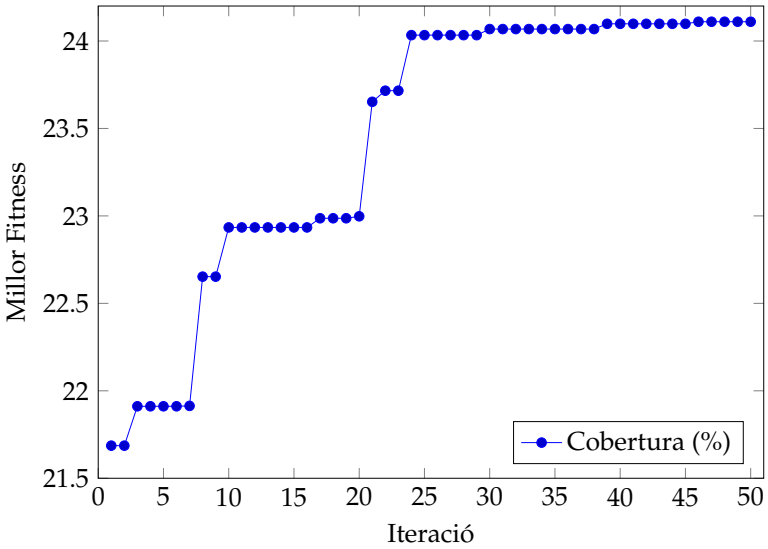


FIGURA 11.8: Evolució del fitness de la millor solució del plànol 1 amb la configuració de la tercera fila de la Taula 11.3

11.2.2 Plànol 2

A la Figura 11.9 es pot veure el mapa de cobertura obtingut amb la configuració a la primera fila de la Taula 11.4 de l'algoritme genètic. Tal com es pot apreciar, és un dels millors possibles, ja que totes les seus estan col·locades de tal manera que cap té la cobertura blocada per cap obstacle. A la Figura 11.10 es pot observar l'evolució del fitness al llarg de les iteracions.

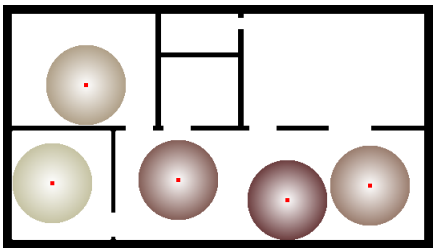


FIGURA 11.9: Mapa de cobertura òptim per 5 serveis amb 50 de radi

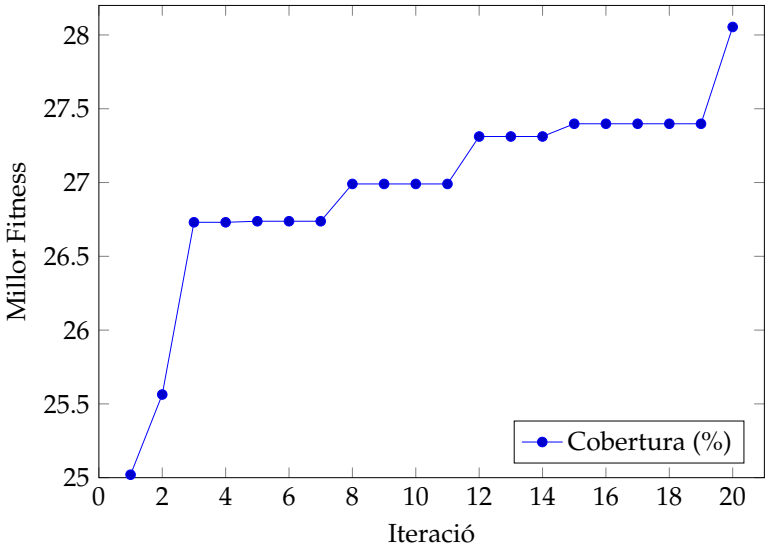


FIGURA 11.10: Evolució del fitness de la millor solució del plànol 2 amb la configuració de la primera fila de la Taula 11.4

A la Figura 11.11 es pot veure el mapa de cobertura obtingut amb la configuració a la segona fila de la Taula 11.4 de l'algoritme genètic. En casos com més complexos com aquest, ja no podem assegurar que la distribució sigui òptima o no, ens hem de conformar amb resultats qualitatius. En aquest cas sembla que la solució podria ser lleugerament millor. A la Figura 11.12 es pot observar l'evolució del fitness al llarg de les iteracions.

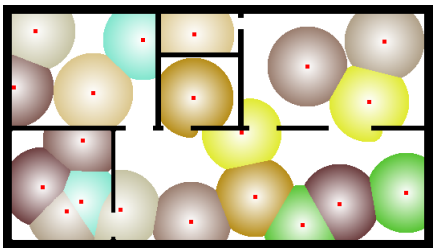


FIGURA 11.11: Mapa de cobertura òptim per 20 serveis amb 50 de radi

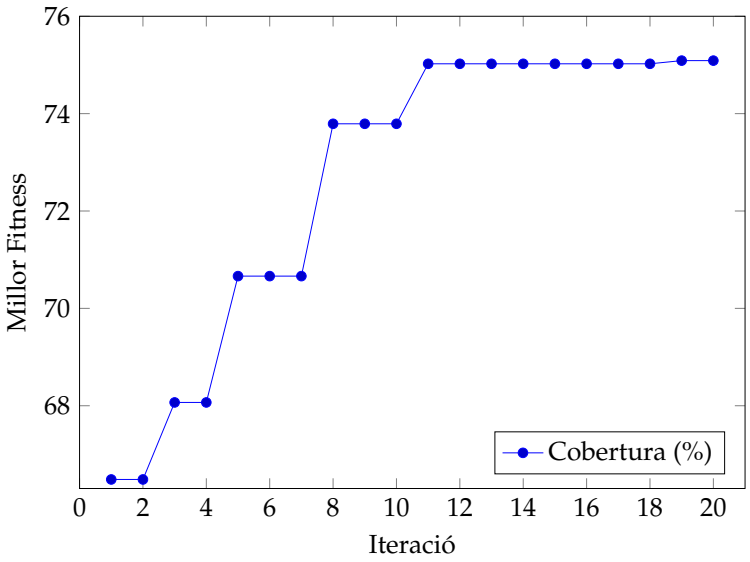


FIGURA 11.12: Evolució del fitness de la millor solució del plànol 2 amb la configuració de la segona fila de la Taula 11.4

A la Figura 11.13 es pot veure el mapa de cobertura obtingut amb la configuració a la tercera fila de la Taula 11.4 de l'algoritme genètic. Tal com es pot apreciar, la solució és bona, però podria ser millor, arribant al 100% de cobertura movent algun dels serveis de la zona inferior esquerra. A la Figura 11.14 es pot observar l'evolució del fitness al llarg de les iteracions.

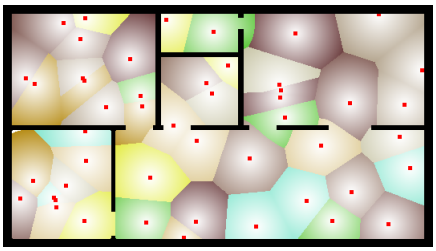


FIGURA 11.13: Mapa de cobertura quasi òptim per 10 serveis amb 105 de radi

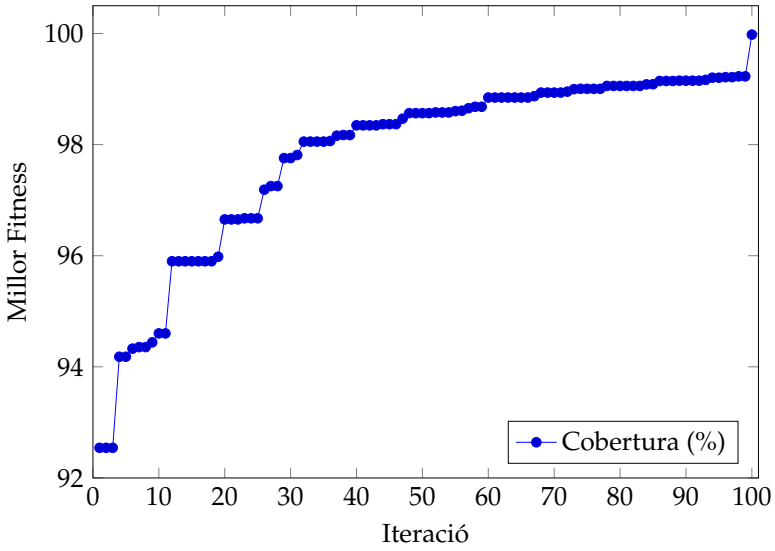


FIGURA 11.14: Evolució del fitness de la millor solució del plànol 2 amb la configuració de la tercera fila de la Taula 11.4

11.2.3 Plànol 3

A la Figura 11.15 es pot veure el mapa de cobertura obtingut amb la configuració a la primera fila de la Taula 11.5 de l'algoritme genètic. Tal com es pot apreciar, sembla que és un dels millors possibles, ja que només una de les seus té la cobertura bloquejada per obstacles. A la Figura 11.16 es pot observar l'evolució del fitness al llarg de les iteracions.

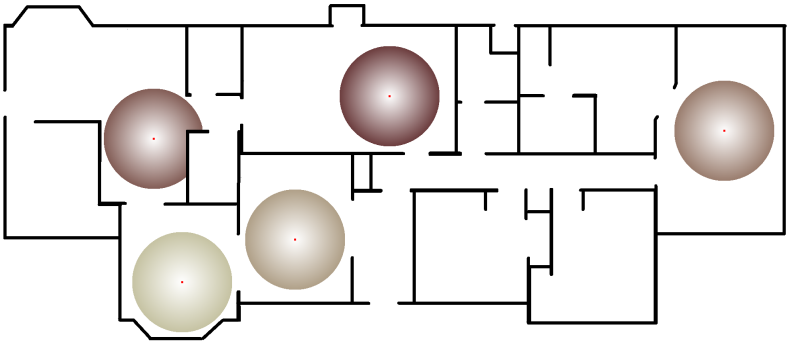


FIGURA 11.15: Mapa de cobertura òptim per 5 serveis amb 125 de radi

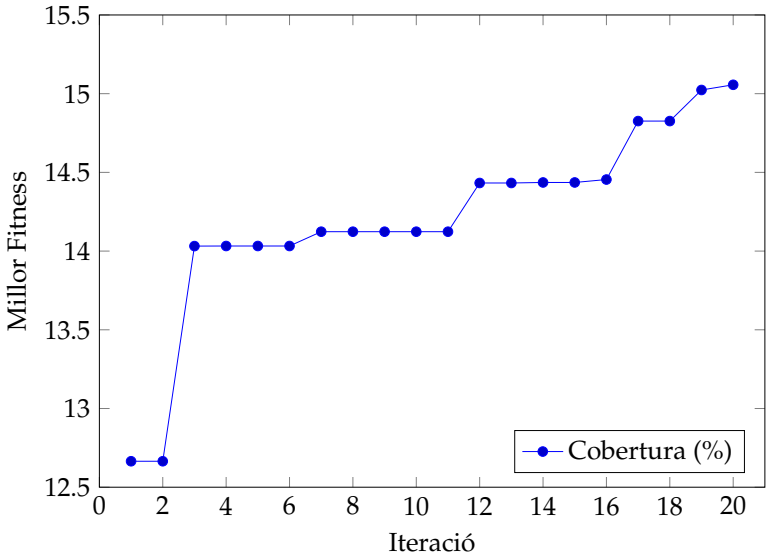


FIGURA 11.16: Evolució del fitness de la millor solució del plànol 3 amb la configuració de la primera fila de la Taula 11.5

A la Figura 11.17 es pot veure el mapa de cobertura obtingut amb la configuració a la segona fila de la Taula 11.5 de l'algoritme genètic. Tal com es pot apreciar, no és un dels millors possibles, ja que els tres serveisque tenen la cobertura bloquejada (part dreta) per obstacles podrien estar millor col·locats. A la **??** es pot observar l'evolució del fitness al llarg de les iteracions.

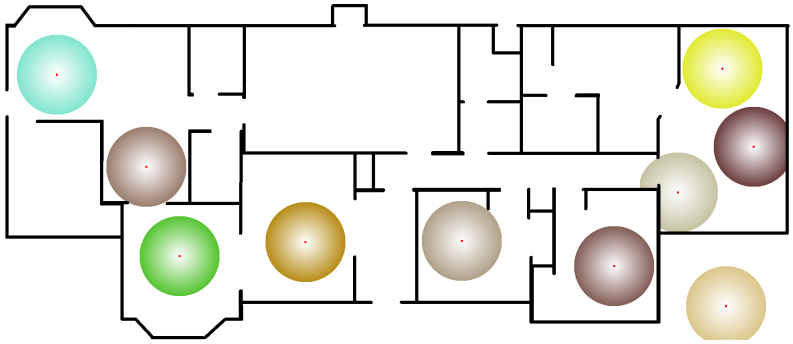


FIGURA 11.17: Mapa de cobertura òptim per 10 serveis amb 100 de radi

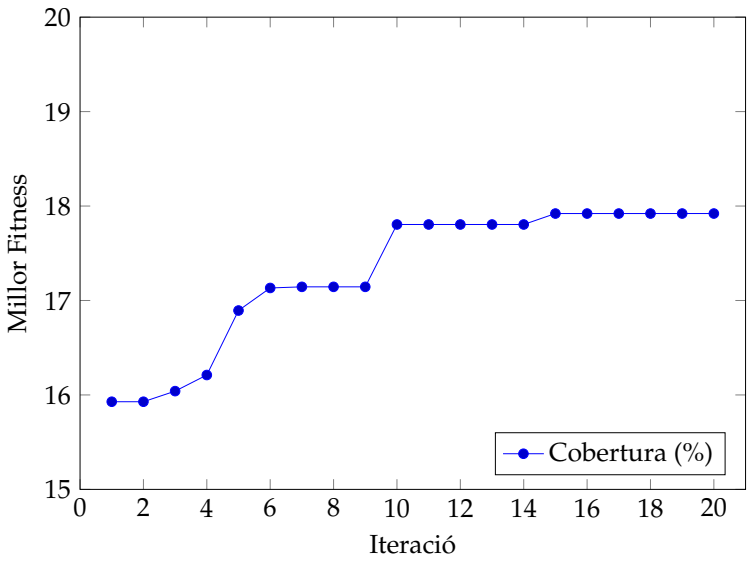


FIGURA 11.18: Evolució del fitness de la millor solució del plànol 3 amb la configuració de la segona fila de la Taula 11.5

A la Figura 11.19 es pot veure el mapa de cobertura obtingut amb la configuració a la tercera fila de la Taula 11.5 de l'algoritme genètic. Tal com es pot apreciar, no és de les millors possibles perquè hi ha múltiples serveis que podrien estar a una millor posició. A la Figura 11.20 es pot observar l'evolució del fitness al llarg de les iteracions.

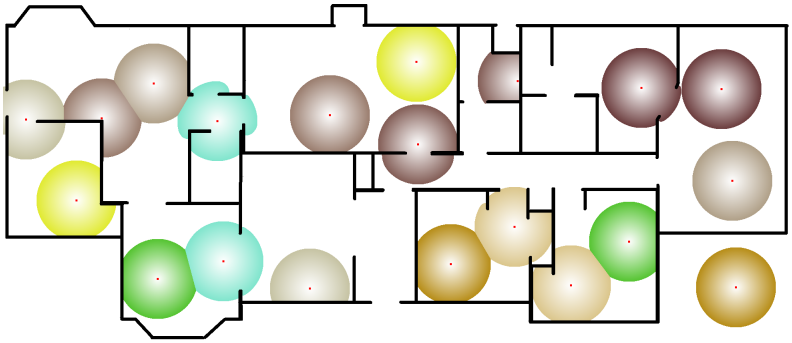


FIGURA 11.19: Mapa de cobertura quasi òptim per 20 serveis amb 50 de radi

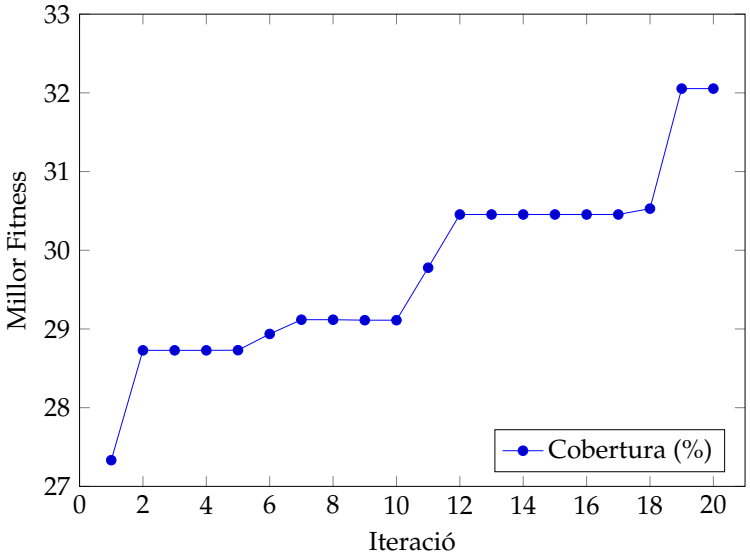


FIGURA 11.20: Evolució del fitness de la millor solució del plànol 3 amb la configuració de la tercera fila de la Taula 11.5

Capítol 12

Conclusions

Els objectius del projecte consistien a desenvolupar una solució per tractar el problema de càlcul de mapes de cobertura utilitzant la capacitat de còmput de les GPU, i emprar-la per trobar solucions òptimes per al problema **MCLP**.

Malgrat que no s'ha pogut acabar de solucionar el problema de les propagacions il·legals, i ha sigut necessari introduir un costós mecanisme addicional per evitar-los, s'han acabat obtenint resultats concordes amb els objectius plantejats, tot i que no s'hagin obtingut de la manera més eficient possible. De totes maneres, tenint en compte el punt de partida del projecte, el caràcter de les complicacions trobades i els resultats que s'ha exposat Capítol 11 es considera que els objectius han sigut assolits satisfactòriament.

Tant en l'àmbit personal com acadèmic he acabat molt satisfet amb el desenvolupament del projecte. He après CUDA des de zero i nous conceptes de C++, que he pogut aplicar a un problema amb aplicacions reals. Alhora he pogut treballar en un entorn d'investigació fabulós que m'ha donat l'oportunitat de contribuir a un projecte de recerca, en el que s'ha elaborat un paper, que ha sigut acceptat al *XX Spanish Meeting on Computational Geometry*, on s'utilitza la solució desenvolupada en aquest projecte.

Addicionalment, aquesta experiència m'ha servit per tenir encara més clar el meu futur acadèmic i professional.

Capítol 13

Treball futur

Com a treball futur hi ha múltiples fronts per tractar.

Des del punt de vista de l'algoritme en si, el més interessant seria descobrir el motiu pel qual es produeixen propagacions il·legals i solucionar el problema d'arrel, sense haver d'incloure mecanismes externs que evitin que l'algoritme s'equivoqui. Alhora estaria bé descobrir perquè es fan més iteracions del compte un cop trobat el mapa de cobertura, que molt probablement està lligat amb el problema anterior.

A més seria força útil, i no molt complicat, adaptar l'algoritme per permetre calcular mapes de cobertura de serveis que travessen obstacles, amb una certa atenuació, com per exemple senyals de wifi.

En termes del sistema seria ideal tenir més nivell de control sobre el càlcul de mapes de cobertura, per exemple acceptant serveis amb diferents radis d'abast. I afegir funcionalitats per millorar l'experiència d'usuari com: poder llegir configuracions a partir de fitxers, guardar els mapes de cobertura en diferents formats, fer la interfície més intuïtiva... Addicionalment, també seria adequat acceptar més formats d'*input*, com imatges vectorials, i afegir la noció d'escala física als dominis per poder utilitzar radis d'abast i donar cobertures en SI, i no utilitzant els píxels com a mètrica, ja que aquests depenen de les dimensions de la imatge.

De totes maneres, probablement l'addició més significativa seria aprofitar els mapes de cobertura obtinguts per resoldre problemes de *path planning*.

Gràcies al fet que els mapes contenen de manera implícita els camins més curts de cada punt cobert fins al servei més proper. No seria difícil agafar un domini, posar un servei a un cert punt objectiu, amb un rang prou gran per cobrir tot el domini, calcular el mapa de cobertura corresponent amb l'algoritme proposat i llavors permetre a l'usuari triar un, o més, punts de partida des dels quals es podria mostrar el camí més curt fins a l'objectiu de manera instantània, sense la necessitat de fer cap càlcul.

Aquesta funcionalitat també seria aplicable a mapes de cobertura arbitraris que no necessàriament cobreixen tot el domini, obrint un ventall de possibles aplicacions.

Capítol 14

Manual d'usuari

El resultat final del projecte és un executable per Windows, disponible al repositori públic de GitHub. L'únic requisit per utilitzar-lo és tenir una targeta gràfica compatible amb CUDA, amb els seus corresponents *drivers* instal·lats.

Un cop executat el programa mostra les opcions de configuració que es poden observar a la Figura 14.1.

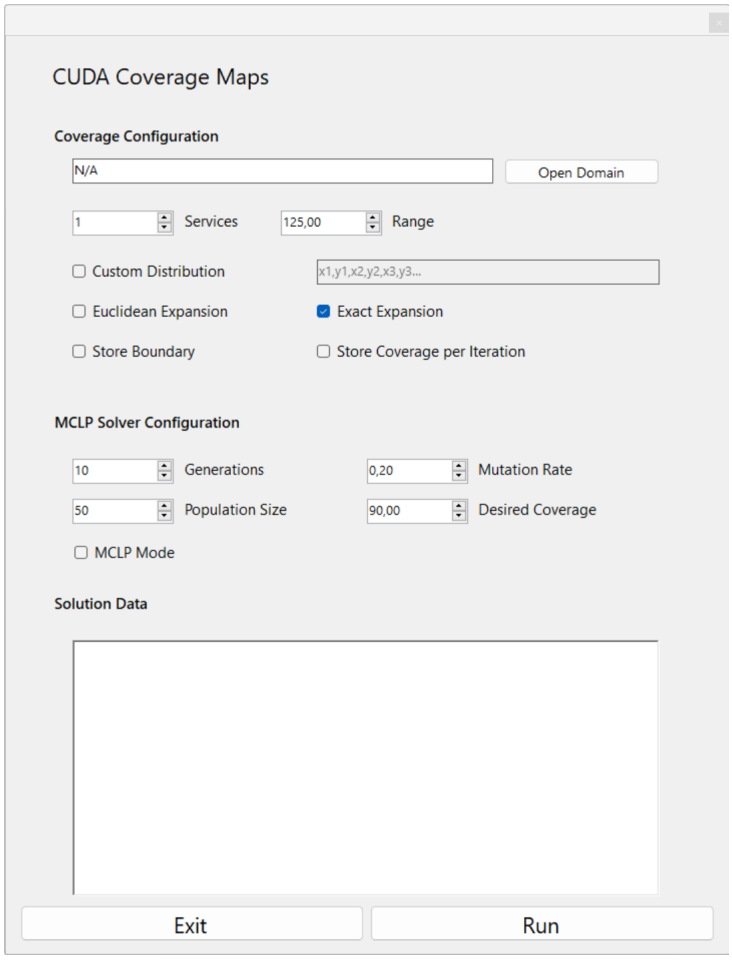


FIGURA 14.1: Pantalla principal del programa

A continuació, es mostra una llista amb el significat de cada un dels elements de la interfície:

- **Open Domain**: botó que al ser clicat obre un explorador de Windows per escollir un domini.
- **Services**: *input* numèric que dicta el nombre de serveis que hi haurà al mapa de cobertura.
- **Range**: *input* numèric que dicta el rang dels serveis.
- **Custom Distribution**: *checkbox* que habilita el requadre de text per entrar distribucions de serveis per text.
- **Euclidean Expansion**: *checkbox* que indica al programa que ha d'utilitzar l'algoritme d'expansió pseudoeuclidiana.
- **Exact Expansion**: *checkbox* que indica al programa que ha d'utilitzar l'algoritme d'expansió pseudoeuclidiana amb correcció de distàncies.
- **Store Boundary**: *checkbox* que indica al programa que ha de guardar el domini en un fitxer `.txt`.
- **Store Coverage per Iteration**: *checkbox* que indica al programa que ha de guardar el mapa de cobertura de cada iteració en un fitxer `.txt`.
- **Generations**: *input* numèric que dicta el nombre d'iteracions de l'algoritme genètic.
- **Population Size**: *input* numèric que dicta la mida de la població de l'algoritme genètic.
- **Mutation Rate**: *input* numèric que dicta la probabilitat de mutació a l'algoritme genètic.
- **Desired Coverage**: *input* numèric que dicta el criteri de parada de l'algoritme genètic.
- **MCLP Mode**: *checkbox* que indica al programa que es vol trobar una solució òptima.
- **Solution Data**: finestra de text on el programa guarda informació relacionada amb la darrera execució.
- **Run**: botó que inicia l'execució.
- **Exit**: botó que surt tanca el programa.

Per seleccionar una imatge només cal clicar l'opció *Open Domain*, que ens presentarà un explorador de Windows en una nova finestra. Un cop ajustats els paràmetres, només cal prémer el botó *Run* i finalitzada l'execució s'actualitza el requadre de text inferior amb dades rellevants d'aquesta, tal com es pot veure a la Figura 14.2, i apareix una nova finestra amb el mapa de cobertura resultant, Figura 14.3.

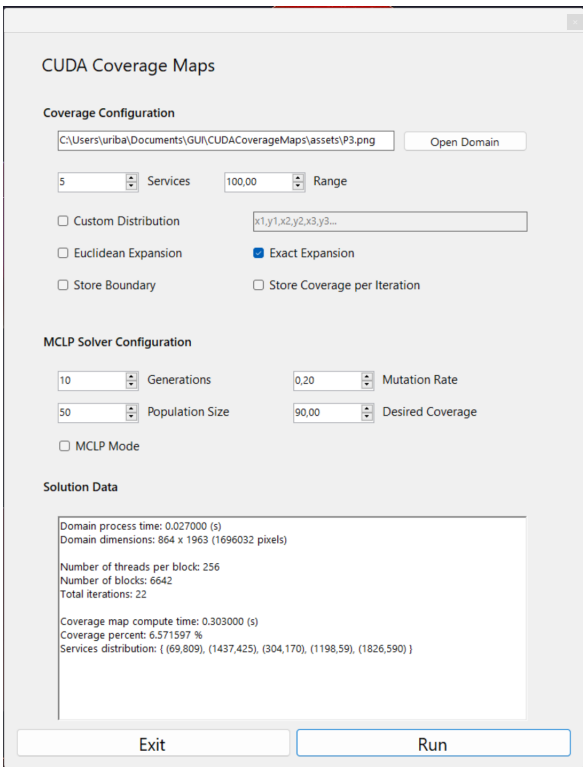


FIGURA 14.2: Pantalla principal del programa un cop ha acabat una execució

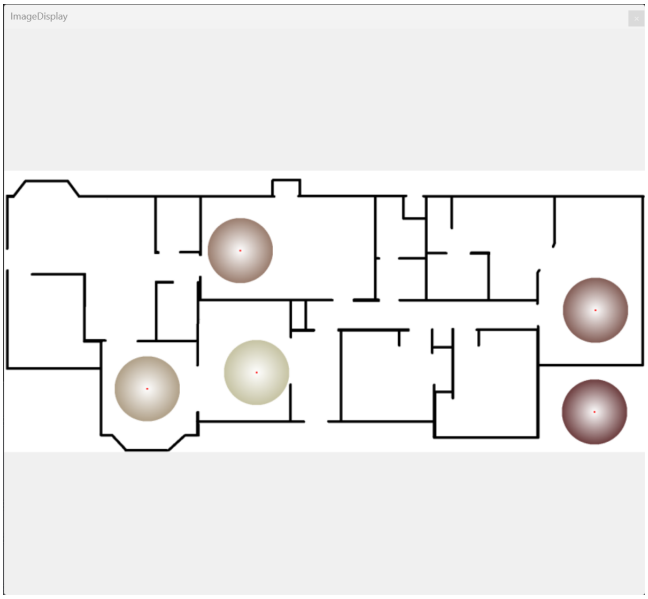


FIGURA 14.3: Finestra que mostra els resultats

Apèndix A

Coverage maps on domains with obstacles. Abstract

XX Spanish Meeting on Computational Geometry, Santiago de Compostela, July 3-5, 2023

Coverage maps on domains with obstacles

Oriol Balló*[1], Narcís Coll†[2], and Marta Fort‡[2]

[1]Universitat de Girona
[2]Graphics and Imaging Laboratory, Universitat de Girona

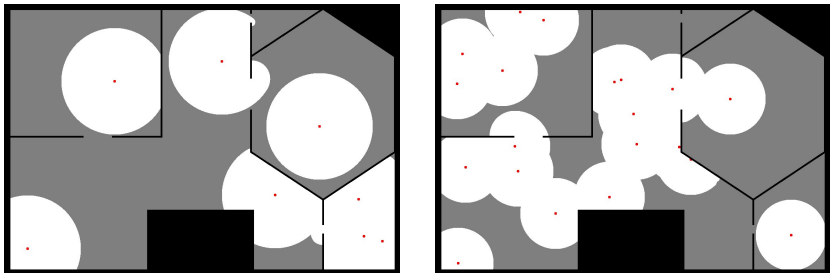


Figure 1: Coverage map of eight facilities (left) and twenty facilities (right)

Abstract

The coverage map of a set of facilities represents, for each point within a domain, whether at least one facility covers it (see Figure 1). That is, we know if at least one facility is at a distance to the point, through free space, smaller than the facility coverage radius. In this work, we propose a parallel method that runs on the GPU to compute coverage maps over domains given by binary images. The method input is an image where each pixel has only one out of two different values depending on whether the pixel represents the free space or not. Meanwhile, the output is an image where each pixel is marked as covered or uncovered by the set of facilities.

We use a two-step iterative process that combines a quasi-Euclidean distance [1] propagation along free space and an exact Euclidean distance computation (without propagation). We iteratively repeat these steps until no updates occur. During the process, we obtain the distance of each pixel to its nearest site and the pixel-*id* of the last corner (of the current shortest path) by using two CUDA-kernels executed on 2d-grids and 2d-blocks and considering a thread per pixel.

The quasi-Euclidean distance propagation along free space uses a GPU parallel Bellman-Ford algorithm. The used graph (computed on the fly from the binary image) has as vertices the free pixels, and connects a free pixel with its, at most 8, neighboring free pixels. Pixels containing sites store a 0 as distance and its pixel-*id* as last-corner-*id*. The rest store ∞ as distance and -1 as last-corner-*id* (uncovered pixel). Similarly to the Euclidean distance transform algorithms [2], each thread analyses its already covered free neighbors and enlarges their path to itself. If the path is shorter than its current distance, it updates its distance and the last-corner-*id* accordingly. An inner-block propagation followed by an inter-block propagation expands the current paths through free space according to the quasi-Euclidean distance. A boolean variable and two synchronizing points (per block) stop the inner-block propagation when no updates occur within the block. The inter-block propagation uses a global boolean to keep calling the kernel while updates occur.

The second step computes the exact Euclidean length of the obtained paths. The covered-pixel threads retrieve the path reaching the pixel while determining its Euclidean length. They add the Euclidean distance from the pixel to its last-corner-*id* to the distance of this last corner to the previous one, and so on, until reaching the original site. This two-step iteration leads to exact free-space coverage maps.

Funding

Research supported by grants PID2019-106426RB-C31 and PDC2021-120997-C32 funded both by MCIN/AEI/10.13039/501100011033 and the 2nd one also by European Union NextGenerationEU/PRTR.

References

[1] Montanari, U. (1968). A Method for Obtaining Skeletons Using a Quasi-Euclidean Distance. Journal of the ACM (JACM), 15, 600–624.

[2] Maurer, C. R., Qi, R., and Raghavan, V. (2003). A linear time algorithm for computing exact Euclidean distance transforms of binary images in arbitrary dimensions. IEEE Transactions on Pattern Analysis and Machine Intelligence, 25(2), 265–270.

*Email: u1962391@campus.udg.edu
†Email: narcis.coll@udg.edu
‡Email: marta.fort@udg.edu

Bibliografia

Ansorge, Richard (31 de maig de 2022). *Programming in Parallel with CUDA: A Practical Guide*. 1a ed. Cambridge University Press. ISBN: 978-1-108-85527-3 978-1-108-47953-0. DOI: 10.1017/9781108855273. URL: https://www.cambridge.org/core/product/identifier/9781108855273/type/book (cons. 27-01-2023).

"NVIDIA Official CUDA C++ Programming Guide" (31 de maig de 2023). A.

Sanders, Jason i Edward Kandrot (2011). *CUDA by example: an introduction to general-purpose GPU programming*. Upper Saddle River, NJ: Addison-Wesley. 290 pàg. ISBN: 978-0-13-138768-3.

Crane, Keenan, Clarisse Weischedel i Max Wardetzky (24 d'oct. de 2017). "The Heat Method for Distance Computation". A: *Communications of the ACM* 60.11, pàg. 90 - 99. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3131280. URL: https://dl.acm.org/doi/10.1145/3131280 (cons. 14-04-2023).

Valero, Alberto (12 de des. de 2013). "Fast Marching Methods in Path Planning". A.

Elizondo-Leal, Juan Carlos, Ezra Federico Parra-González i José Gabriel Ramírez-Torres (1 de juny de 2013). "The Exact Euclidean Distance Transform: A New Algorithm for Universal Path Planning". A: *International Journal of Advanced Robotic Systems* 10.6, pàg. 266. ISSN: 1729-8814, 1729-8814. DOI: 10.5772/56581. URL: http://journals.sagepub.com/doi/10.5772/56581 (cons. 27-05-2023).

R.L. Church i C. ReVelle (1974). "The Maximal Covering Location Problem". A: *Papers of the Regional Science Association* 32, pàg. 101 - 118.

R.L. Church (1984). "The planar maximal covering location problem". A: *Journal of Regional Science* 24.2, pàg. 185 - 201.

N. Megiddo, E. Zemel i S.L. Hakimi (1983). "The maximum coverage location problem". A: *SIAM Journal of Algebraic and Discrete Methods* 4.2, pàg. 253 - 261.

Hochbaum, Dorit S. (1996). "Approximating Covering and Packing Problems: Set Cover, Vertex Cover, Independent Set, and Related Problems". A: *Approximation Algorithms for NP-Hard Problems*. USA: PWS Publishing Co., pàg. 94 - 143. ISBN: 0534949681.

Poli, Riccardo et al. (2008). *A Field Guide to Genetic Programming*. [Morrisville, NC: Lulu Press]. ISBN: 978-1-4092-0073-4.

Koza, John R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Complex Adaptive Systems. Cambridge, Mass: MIT Press. ISBN: 978-0-262-11170-6.