als que m'aguanten
als que no ho fan
i especialment a L, G, M, I i Z

i gràcies a Ariadna per la correcció

Índex

Índex de taules

Índex de figures

1. Introducció

1.1 Motivació

La resolució de problemes mitjançant algorismes d'intel·ligència artificial és un estudi fascinant. Analitzar el problema, trobar una bona representació i trobar les tècniques adients per afrontar-lo és un procés extens i que planteja tot un seguit de reptes molt variats i didàctics, així que vam cercar un problema que fos prou accessible i alhora complicat com per què resultés un bon mitjà d'aprofundir en els mètodes i sistemes de la intel·ligència artificial.

Els jocs constitueixen un problema clàssic d'intel·ligència artificial. A part del seu interès propi, també poden plantejar punts en comú amb problemes reals i són un bon punt de partida per experimentar amb algorismes de resolució de problemes. Els algorismes de cerca en arbres són una de les tècniques més immediates per resoldre jocs.

Fent servir una representació en arbre, on els nodes representen els estats del joc i les transicions entre estats representen els possibles moviments del joc, podem partir dels algorismes clàssics de cerca en arbres (amplada prioritària, profunditat prioritària...) per trobar solucions, és a dir, els moviments necessaris per arribar a un estat final a partir de l'estat inicial, o bé en el cas dels jocs amb adversari, trobar el millor moviment en un estat donat. Els arbres de joc amb adversari tenen peculiaritats que els distingeixen dels jocs unipersonals i per resoldre'ls es fan servir algorismes diferents, com ara el popular Minimax.

Normalment, a aquests algorismes de cerca hi afegim informació sobre el domini del problema, com ara funcions heurístiques que ens avaluen els nodes de l'arbre i ens 'guien' en la cerca, o podes de l'arbre que ens eliminen nodes dolents de l'arbre.

Més enllà dels algorismes de cerca, existeixen altres possibilitats per resoldre jocs, com per exemple els mètodes de planificació. Aquests mètodes intenten trobar una seqüència de tasques que resolen una tasca general més gran i poden ser aplicables a jocs unipersonals.

El següent pas era trobar un joc que fos suficientment interessant i que ens proporcionés problemes de dificultat variable. Vam escollir un joc unipersonal moderadament popular, el Sokoban.

1.2 Què és Sokoban?

El Sokoban (Sokoban © 1982 Thinking Rabbit Inc. Japan), el joc de l'home del magatzem japonès, va ser inventat l'any 1982 per la companyia japonesa Thinking Rabbit.

El joc es desenvolupa sobre una quadrícula de dimensió arbitrària que representa un magatzem, amb quadres per on es pot passar i parets. Inicialment, hi han quadres accessibles ocupats per unes caixes, i uns altres que són els quadres objectiu. També hi ha l'home del magatzem. Podem veure un exemple a la figura 1.1

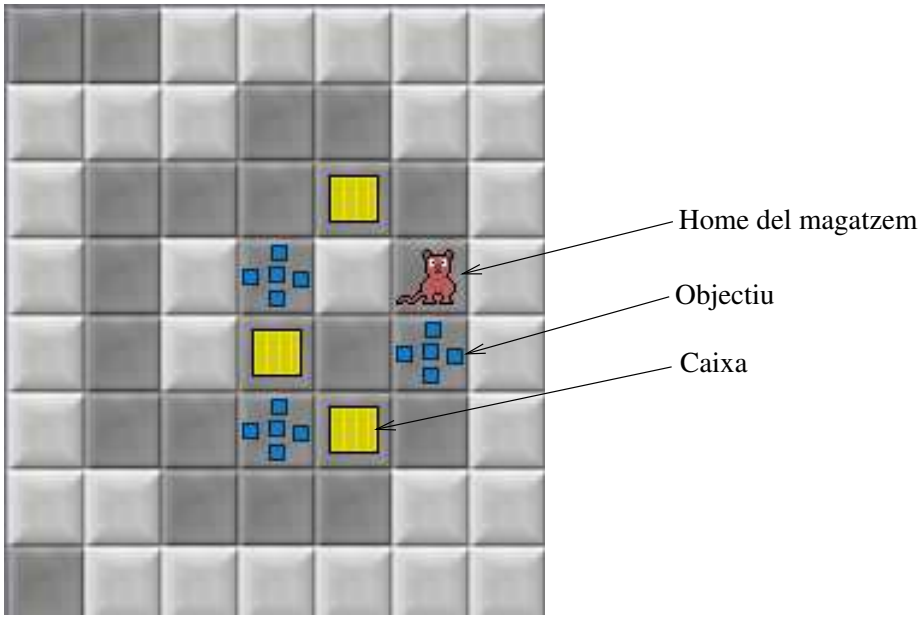

Figura 1.1: Exemple de mapa

L'home del magatzem és el nostre jugador. Es pot desplaçar en les quatre direccions (d'ara endavant, nord, oest, est i sud), sempre que no tingui un

obstacle. L'home del magatzem pot trepitjar els quadres objectiu.

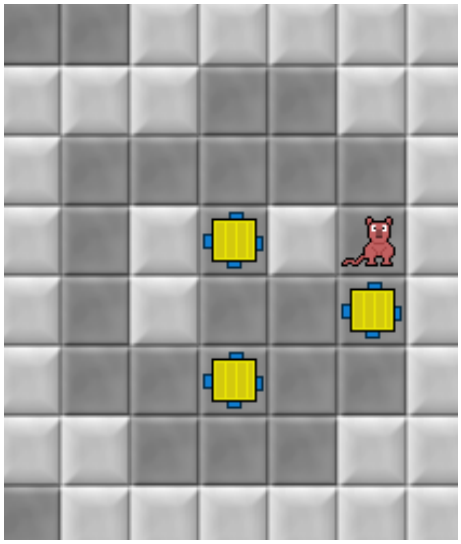

Figura 1.2: Exemple de mapa resolt

L'objectiu de l'home del magatzem és portar les caixes als quadres objectiu (com a la figura 1.2), movent-les mitjançant empentes. Per empènyer una caixa cap a l'est, per exemple, l'home s'ha de situar a la casella a l'oest de la caixa i la casella a l'est de la caixa ha d'estar buida (o ser un quadre objectiu). Si l'home es mou cap a l'est, la caixa també es desplaçarà cap a l'est (figura 1.3).

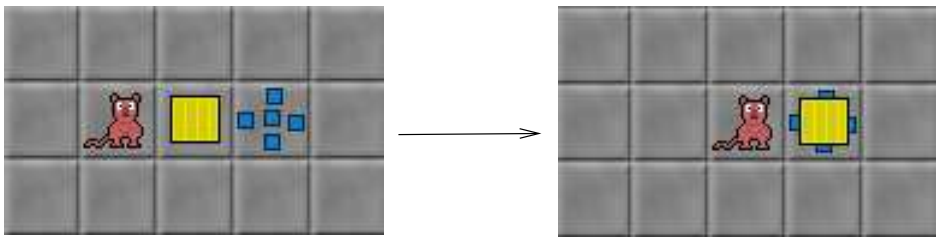

Figura 1.3: Empenyent un bloc

No hi ha més alternatives:

- L'home no pot saltar per sobre dels obstacles
- No pot empènyer dos o més caixes alhora.

- No es permeten moviments en diagonal

És interessant ressaltar que una caixa pot quedar atrapada sense poder moure-la enlloc; o inclús pot ser mòbil, però quedar en una posició en la qual mai pot arribar a una casella objectiu. A la figura 1.4 veiem un bloc totalment immòbil i un altre, que tot i que encara es pot moure, mai no pot arribar al seu objectiu.

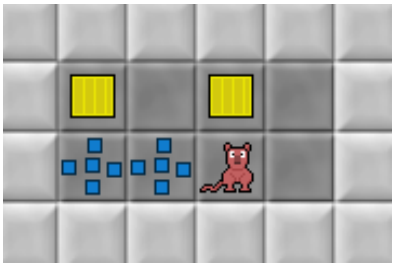

Figura 1.4: Exemple de blocs encallats

També pot ser que aquesta situació passi exclusivament mitjançant interaccions de blocs, com a la figura 1.5.

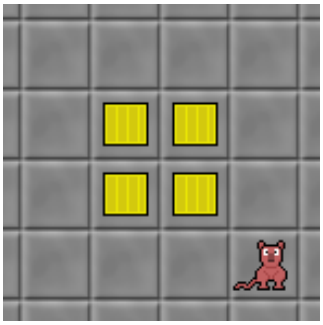

Figura 1.5: Blocs encallats entre ells

Si arribem a una d'aquestes configuracions, haurem arribat a un cul-de-sac i haurem de desfer alguns moviments, doncs des d'una d'aquestes posicions, els mapes ja no es poden resoldre.

1.3 Per què Sokoban?

El joc del Sokoban té tot un seguit de característiques que el fan particularment interessant:

- Les regles són extremadament senzilles i s'entenen en un moment.
- Existeixen multitud de nivells creats, amb una immensa varietat de problemes diferents i estratègies necessàries per la resolució.
- El Sokoban planteja situacions cul-de-sac en les que s'arriba a un estat que no es pot resoldre.
- És un problema que té certes similituds amb els problemes clàssics de planificació.

De cara a la seva resolució algorísmica, el joc del Sokoban es considera extremadament dur de resoldre amb un ordinador. El programa més fort de Sokoban, Rolling Stone[4] resol només 52 dels 90 nivells de l'XSokoban (nivells que són el *benchmark* més comú), quan tots ells han estat resolts per humans. S'ha demostrat que és un problema NP-hard i PSPACE-complet[1] (l'espai requerit per resoldre'l és polinomial i altres problemes PSPACE es poden reduir a problemes de Sokoban).

1.4 Objectius

El nostre objectiu és dur a terme el procés complet d'analitzar i resoldre un problema d'intel·ligència artificial. Aplicarem algorismes de cerca de cara a intentar resoldre mapes de Sokoban i desenvoluparem una aplicació que ens permeti avaluar aquests algorismes i analitzar l'impacte de les millores i variacions que intentem aplicar.

Alhora desitgem que aquesta aplicació sigui molt accessible, tant per un usuari que desitgi executar-lo com per un usuari que vulgui estendre'l i modificar-lo.

1.5 Organització de la memòria

Després d'aquesta introducció, en el capítol 2 veurem una descripció dels algorismes de cerca que aplicarem al problema i un recull dels *solvers* existents de Sokoban. Al capítol 3, veurem com apliquem aquestes tècniques al joc del Sokoban i quines millores hi hem aplicat. El capítol 4 conté un anàlisi a nivell tècnic de la implementació. Després farem una anàlisi del rendiment d'aquesta al capítol 5. Finalment, en el capítol 6 veurem el resultat del procés i n'analitzarem els resultats.

Els apèndixs contenen una explicació dels recursos que hem fet servir (apèndix A) i una referència del mode batch del programa (apèndix B).

2. Fonaments i treballs anteriors

Dins del procés de treballar amb el problema del Sokoban hem de ser conscients que ja existeix tot un seguit de coneixements que podem aplicar. Aquests coneixements són tant genèrics de resolució de problemes, que podem aplicar al nostre joc; com treball existent sobre el joc del Sokoban.

2.1 Algorismes de cerca

Com ja hem vist, els algorismes de cerca són molt aplicables als jocs unipersonals com Sokoban.

Definim un joc típic com una sèrie d'estats que componen les possibles posicions de joc, distingint normalment un estat inicial. Els estats estan connectats per transicions que equivalen als moviments legals del joc. L'objectiu generalment es trobar la combinació de moviments que ens porten des de l'estat inicial fins a una solució.

Sembla immediat fer servir un arbre en el qual els nodes són estats de joc i les transicions, moviments. Així, cada node tindrà com a nodes fills les possibles posicions derivades d'ell.

En molts jocs, desitgem que la solució que trobem tingui el menor nombre de moviments possible. Hi ha jocs, que no analitzarem aquí, en els quals es desitjable optimitzar segons altres criteris, amb els quals és convenient aplicar refinaments addicionals com assignar costos a les transicions, i on s'apliquen algorismes modificats per tenir en compte aquests criteris.

Els algorismes de cerca recorren aquest arbre fins a trobar un estat final; la principal diferència entre els diferents mètodes de cerca és en quin ordre la fan. Això no només afecta el nombre de nodes que hem d'analitzar abans de trobar la solució (i per tant el temps de resolució), sinó que a més a més,

pot fer que trobem solucions diferents.

Els algorismes bàsics de cerca estableixen 2 ordres, la *profunditat prioritària*, que sempre intenta descendir l'arbre fins que troba la solució o ha de tornar enrere, i l'amplada prioritària, que analitza els nodes en ordre de profunditat de l'arbre; primer els de profunditat 1, després 2 i així successivament.

Destaquem que la profunditat prioritària no és pràctica si tenim camins infinits (doncs l'algorisme pot decidir baixar per un camí infinit) i que la profunditat prioritària ens assegura que la solució que trobarem és de profunditat mínima (i, per tant, generalment òptima).

Tot i que aquests algorismes de cerca funcionen, la majoria de jocs requereixen molt treball per resoldre'ls així. Si suposem un joc senzill on en cada moment només existeixen dues alternatives, veiem fàcilment que si la solució es troba a profunditat 34 necessitem examinar $2^{35} = 34359738368$ posicions de joc amb amplada prioritària per trobar la solució. Suposant que fos possible analitzar un milió de posicions per segon, trigaríem unes 9 hores en trobar la solució.

Per intentar reduir aquestes quantitats, hem de fer servir tècniques específiques per cada problema. Aplicant coneixements que ens permeten jutjar la qualitat d'un moviment o d'una posició podem aconseguir 'podar' l'arbre i aplicar la cerca en un ordre més adient.

2.1.1 Cerques heurístiques

Un dels primers conceptes específics del problema que s'introdueix en les cerques d'arbres és la funció heurística. Una funció heurística valora la proximitat de cada node a la solució. Òbviament intentarem analitzar abans els nodes que tinguin una bona valoració heurística.

Depenent de l'ús que fem servir de la funció heurística obtindrem diferents algorismes de cerca heurística

Si decidim analitzar sempre el node de millor heurística, estarem realitzant una cerca *best-first*. Si l'heurística és bona, la cerca *best-first* ens portarà més ràpidament a una solució que una cerca per amplada prioritària, però per contra, no està garantit que la solució que trobem sigui la que es troba a menys profunditat.

L'algorisme A* intenta fer servir l'heurística per orientar la cerca però intentant optimitzar alhora la resolució. Afegirem una funció $g(x)$ que ens diu el cost que té un estat de joc x i definirem la nostra funció heurística $h(x)$ com una estimació del cost que ens queda per arribar a la solució.

Amb això podem considerar la funció $f(x) = g(x) + h(x)$, que ve a representar el cost estimat de la solució que trobaríem en aquest camí de resolució.

Si analitzem primer els nodes de menor $f(x)$ veurem que estem fent una cerca guiada per l'heurística però que també intenta trobar una solució de mínim cost. De fet, podem veure que si $h(x)$ és una cota inferior del cost real que ens queda per arribar a la solució, el mètode A* ens portarà a una solució òptima. Les heurístiques que compleixen aquesta condició es denominen **heurístiques admissibles**.

Podem intuir que un criteri per comparar heurístiques és, simplement, comparar els valors numèrics. Entre dues heurístiques admissibles, la que doni valors més grans serà millor. Aquí s'introdueix el concepte d'heurística dominant; una heurística h_1 domina l'heurística h_2 si per a cada posició x possible, $h_1(x) \geq h_2(x)$. Donat un conjunt d'heurístiques $h_1, h_2, \ldots, h_n$, podem aconseguir una heurística dominant $h_d = \max(h_1, h_2, \ldots, h_n)$.

2.1.2 Cerques òptimes i bàsiques

Veiem que amb diferents algorismes de cerca obtenim diferents solucions. Això ens planteja l'elecció de si volem que la nostra solució compleixi algun criteri d'optimalitat. En moltes situacions, es desitjable que la solució sigui òptima sobre un cost, típicament el nombre de moviments (com veurem, en el cas del Sokoban existeixen diversos tipus d'optimalitat). Ara bé, sembla clar que això tindrà el seu cost en quant a temps de resolució.

Efectivament, això és cert per la majoria de jocs. Intuïtivament, fer una cerca heurística en general fa que es necessiti recórrer menys nodes per arribar a la solució, però potser dins dels nodes que ens 'estalviem' es trobaria el camí a una solució òptima. L'A* ens permet aprofitar una heurística i obtenir solucions òptimes.

2.2 Solvers existents

Encara que Sokoban no és tant popular com ho poden ser els escacs o altres jocs clàssics en intel·ligència artificial, existeixen diversos projectes sobre Sokoban, tant de resolució automàtica com d'altres àrees (generació automàtica de mapes, per exemple). Molts d'aquests projectes són japonesos, així que només destacarem un parell dels quals existeix informació en anglès i francès.

2.2.1 Solver Takaken

El *solver* Takaken és força conegut en el món del Sokoban, tot i que està força limitat a puzles senzills. N'existeixen una pila de variants amb diverses mi-

llores. Ens centrarem en la versió de Paul Voyer, doncs està documentada en francès (la documentació i informació del Takaken original està en japonès), tot i que conserva parts del codi font en japonès.

La versió de Paul Voyer limita els puzles a laberints de 18x14 i 11 blocs.

El *solver* de Takaken fa una cerca exhaustiva de l'arbre d'empentes de blocs (és a dir, no es preocupa dels moviments de l'home fins que no toca mostrar la solució completa) fent servir taules hash i cerques binàries en certs punts per accelerar l'algorisme.

2.2.2 Rolling Stone

Rolling Stone és possiblement el millor programa per resoldre Sokoban que existeix. Escrit en C++, es basa en l'algorisme IDA* (Iterative Deepening A*), una millora de l'algorisme A*. Rolling Stone pretén demostrar que la cerca d'agent únic és insuficient per resoldre problemes complexos.

Segons els seus autors, Rolling Stone no resol **cap** dels nivells d'XSokoban només aplicant l'IDA*. A partir d'això, s'ha anat incorporant millores a Rolling Stone que augmenten el seu rendiment:

Minmatching Una sofisticada cota inferior pel nombre de moviments de bloc necessaris per resoldre una configuració arbitrària del nivell. Fa servir un algorisme que resol el match perfecte de cost mínim en grafs bipartits [4].

Taules hash Es fan servir per eliminar posicions repetides en l'arbre i bucles. [4].

Ordenació de moviments S'ordenen els moviments abans de fer la cerca. Els beneficis només són esperats en l'última iteració, però poden ser substancials [4].

Taules de Deadlock Una cerca de patrons per detectar posicions irresolubles en regions de 5x4 [4].

Macros de túnel Es considera el moviment d'un bloc a través d'un túnel com un únic moviment [4].

Macros d'objectiu Un cop un bloc entra en una regió de caselles objectiu, se'l porta automàticament a l'objectiu adient [4].

Cuts d'objectiu S'assumeix que el macro anterior és l'únic moviment correcte [4].

Cerques de patrons En nodes que es consideren 'perillosos' (heurísticament), l'IDA* invoca una cerca especulativa que intenta identificar conflictes entre subconjunts limitats de blocs que, posteriorment, seran fets servits en la cerca per millorar la cota inferior [6].

Cuts de rellevància Es desestimen moviments no relacionats amb l'anterior [5], [3].

Sobreestimació En comptes de fer servir una cota inferior com estimació heurística de la distància a la solució, s'utilitza una estimació realista que de vegades sobreestima [7].

Reinici ràpid aleatori Quan es creu que la cerca està encallada, es reinicia i es reordenen els moviments. Més reordenament en cada intent, fins 8 cops per iteració d'IDA*. Llavors s'assumeix que no hi ha solució per aquest llindar i s'incrementa.

2.2.3 Discussió

Actualment sembla que el desenvolupament sobre Sokoban està estancat. A part de la resolució -on encara no s'ha arribat al nivell humà de resolució- hi ha més àrees de recerca possibles sobre Sokoban; s'han fet treballs sobre generació automàtica de mapes de Sokoban, per exemple, i existeixen variants del problema que poden plantejar nous reptes (Sokoban amb diversos homes del magatzem, geometries alternatives, etc.).

3. El Sokoban com a problema de cerca en arbre

Així doncs, intentarem aplicar els algorismes de cerca en arbre per tal de trobar solucions a puzles de Sokoban. Per començar, ens cal definir l'arbre en el qual farem cerques.

3.1 L'arbre

Clarament, els nodes del nostre arbre seran estats de joc; un node serà el resultat d'aplicar una sèrie de moviments de l'home del magatzem sobre la posició inicial. Ara bé, ¿com construïm aquest arbre?

La primera manera natural que se'ns acudeix és definir que d'un node es deriven fins a quatre nodes fills: moure l'home cap al nord, sud, est i oest. Tenim un exemple a la figura 3.1.

La classe `SituacioSimple` és la nostra primera implementació d'aquesta idea. El mapa es representa com un array bidimensional que conté les caselles del mapa.

Considerem els moviments de l'home del magatzem com les possibles continuacions a aquest moviment, és a dir, que durant l'expansió, afegirem les quatre possibles situacions que es creen a partir de moure l'home al nord, sud, oest o est (si són moviments vàlids de joc).

Òbviament, qualsevol solució vàlida d'un puzle estarà en aquest arbre, així que en principi ens hauria de servir per resoldre qualsevol mapa. Però, podem aconseguir una representació millor?

Si analitzem la solució de qualsevol mapa, veurem que podem descomposar la seqüència de moviments en una repetició:

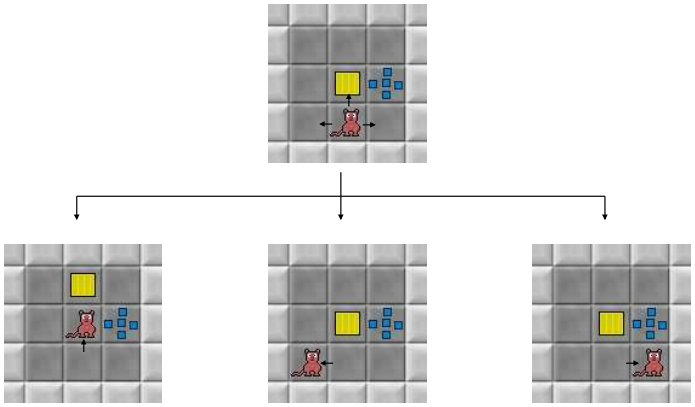

Figura 3.1: Expandint nodes pels moviments de l'home

- Moure's cap a la casella veïna d'una caixa
- Empènyer la caixa

Així doncs, podem considerar com 'moviment', tota una seqüència de moviments d'home, en els quals l'últim (i cap més), és una empenta de bloc. De fet, podem establir que la seqüència de moviments que ens porta fins la casella des d'on empenyarem el bloc sigui el camí òptim, eliminant un munt de possibilitats que no són òptimes. Podem veure un exemple d'això a la figura 3.2.

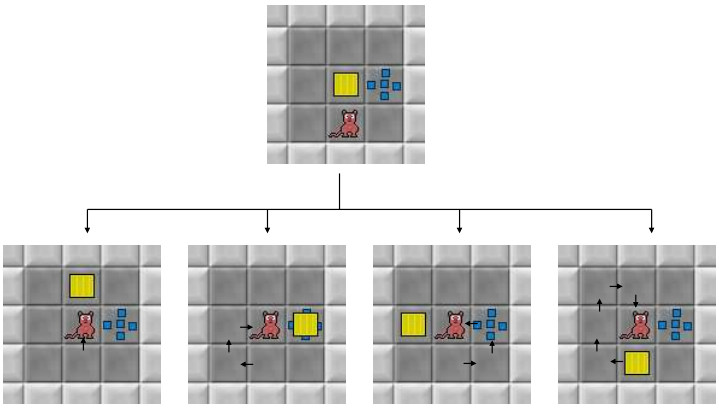

Figura 3.2: Exemple per `SituacioComplexa`

L'arbre que resulta és força diferent al que obtenim amb els moviments d'home. Veiem que el nombre de fills possibles d'un node s'incrementa; passem d'un màxim de 4 nodes a un màxim de 4 cops el nombre de caixes dins

el mapa. La profunditat de l'arbre, per contra, disminueix. El nombre d'empentes de bloc necessaris per arribar a una situació donada és sempre inferior al nombre de moviments d'home necessaris, i a la pràctica, molt inferior. Més endavant veurem si la reducció de profunditat compensa la major ramificació de l'arbre.

L'expansió es fa amb un algorisme senzill, en la classe `SituacioComplexa`. Primer construïm un mapa de camins que ens detalla la ruta òptima de l'home fins a cada casella accessible del mapa. Després, per cada bloc, comprovem si els moviments des de caselles veïnes constitueixen empentes vàlides. Les empentes des de caselles accessibles constitueixen els successors del node.

3.2 L'algorisme de cerca

Vam implementar un senzill algorisme que es pot configurar per comportar-se com una cerca d'amplada prioritària, *best-first* i A*. L'algorisme es basa en dues llistes; una llista de nodes a analitzar i una llista de nodes analitzats. Inicialment, posem la posició inicial del mapa en la llista de nodes a analitzar.

El procés iteratiu és molt senzill: extraiem un node de la llista de posicions a analitzar, si es tracta d'una posició final del joc, ja hem acabat. Si no l'és, el posem a la llista d'analitzats i posem les posicions derivades d'aquest node dins de la llista de nodes a analitzar.

Canviant el criteri per treure el node a examinar de la llista de posicions a analitzar, podem aconseguir que aquest algorisme es comporti com diferents algorismes de cerca:

- Si agafem el node de menor profunditat de l'arbre, estarem fent una cerca en amplada prioritària.
- Si agafem el node amb millor heurística, estarem fent una cerca *best-first*.
- Si agafem el node tal que el seu valor heurístic més el seu cost acumulat sigui mínim, estarem fent un A*.

El segon mecanisme que ens permet variar el mecanisme de resolució és l'elecció de la representació de les posicions i l'algorisme d'expansió. Definim la interfície `Situacio`, que inclou aquests punts; així podrem experimentar amb diferents implementacions de `Situacio`.

3.3 Heurística inicial

La nostra heurística inicial és extremadament senzilla. Agafem cada caixa i calculem la distància de Manhattan fins l'objectiu més proper; la nostra heurística és la suma de totes les distàncies.

A la figura 3.3 podem veure el cost que s'assignaria a un bloc per cada casella del mapa.

Donat que cada moviment de l'home del magatzem com a màxim redueix en 1 la distància d'un bloc a l'objectiu al qual anirà a parar, és fàcil veure que aquesta mesura és una cota mínima del nombre de moviments que queden per resoldre el mapa i del nombre d'empentes, fent que sigui vàlida tant per `SituacioSimple` com per `SituacioComplexa`.

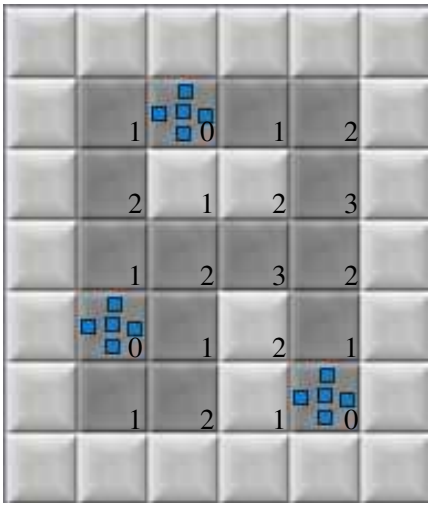

Figura 3.3: Exemple pel càlcul de heurística

Tot i això, és fàcil veure que aquesta heurística presenta molts problemes:

1. L'heurística 'considera' que dues caixes poden anar al mateix objectiu. Això pot ser un problema greu per a certs tipus de mapes, inclús de molt simples.

2. Està molt lluny de ser una estimació propera al nombre de moviments que ens queden. Per posar un petit exemple, en el nostre mapa de prova principal, l'heurística inicial és 5, quan es necessiten 90 moviments per resoldre'l.

3. Molts mapes requereixen allunyar les caixes dels objectius.

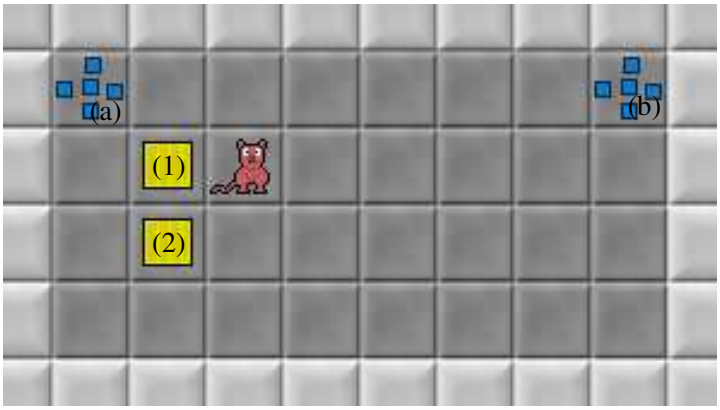

Figura 3.4: Exemple de problemes amb l'heurística

Podem veure un senzill exemple del primer problema en la figura 3.4. Un jugador humà veu clarament que la solució és portar el bloc (1) a l'objectiu (a) i el (2) al (b). Ara bé, portar un bloc cap a (b) ens comporta una reducció d'heurística; la distància cap a l'objectiu més proper augmenta. Això fa que el algorisme s''oposi' a aquest moviment (i.e.: no el considerarà fins que no hagi desistit de portar tots dos blocs cap al mateix objectiu).

Aquest problema es pot resoldre. Rolling Stone fa servir[4] un algorisme de matching de grafs bipartits per trobar una parella objectiu per a cada caixa, de manera que la suma de les distàncies de cada bloc al seu objectiu sigui mínima.

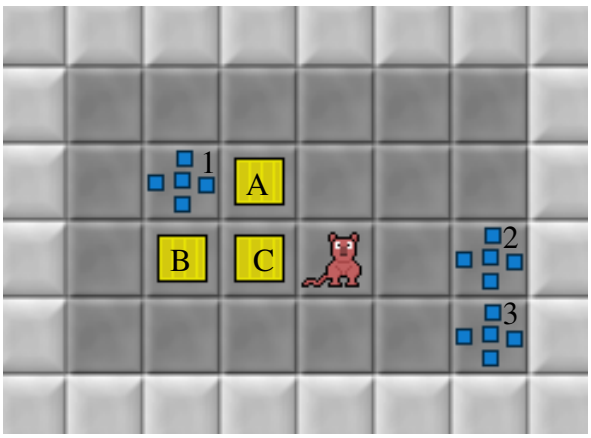

Figura 3.5: Mapa d'exemple pel matching de grafs bipartits

Per exemple, al mapa de la figura 3.5, podem construir la taula de

distàncies (considerant la distància com la de Manhattan, per exemple) de la taula 3.1.

	A	B	C
1	1	1	2
2	4	4	3
3	5	5	4

Taula 3.1: Taula de distàncies pel matching de grafs bipartits

Un possible matching seria A1, B3, C2, que ens donaria un cost total de 9. Existeixen algorismes[2] que ens troben el millor matching possible.

Tot i que aquest matching possiblement no ens associarà cada caixa amb l'objectiu al qual anirà a parar finalment, proporciona una millora sensible, sobretot tenint en compte que existeixen algorismes que calculen aquest matching ràpidament. Òbviament, això manté l'admissibilitat de l'heurística, doncs aquest valor serà, per definició, més petit que el cost de portar cada caixa a un objectiu.

La distància de Manhattan, com veurem més endavant, tampoc no és la millor distància que podem fer.

Es fa difícil trobar més millores a l'heurística que compleixin els requisits de l'heurística de l'A*. De fet, els programadors de Rolling Stone no fan servir una heurística admissible[7], però nosaltres mantindrem l'admissibilitat com a requeriment. Donat que els valors heurístics típicament seran molt petits en relació al nombre de moviments requerits, la influència de l'heurística en la resolució del problema serà petita (això ho comprovarem experimentalment en l'avaluació de casos concrets; veurem que no hi ha massa diferència entre l'A* i una cerca en amplada prioritària).

3.4 Millores aplicades

Amb la representació i l'heurística n'hi ha prou per resoldre un grapat de mapes de Sokoban. Tot i això, en el nostre conjunt de mapes de prova encara hi ha molts que no es resolen.

Veiem que és necessari aplicar quelcom més per aconseguir un programa millor. Les millores que considerarem són de diversos tipus:

Millores específiques Es tracta de millores que tenen a veure amb el joc del Sokoban. Per exemple, veurem més endavant una poda de l'arbre

que elimina posicions irresolubles amb un criteri molt senzill (que no hi hagin blocs encallats)

Millores genèriques Millores a l'algorisme genèric de resolució. Un exemple seria mantenir la llista de posicions examinades, que ens evita fer feina innecessària.

Millores de fons Considerem com una millora de fons canvis en el programa i en el sistema que no afecten el mètode de resolució, però que tenen un impacte en el temps de resolució, com per exemple escollir estructures de dades més adients.

A l'hora d'avaluar millores hem de trobar un criteri adequat. És força temptador escollir el temps de resolució com a mesura de la velocitat del nostre programa. Ara bé, si l'escollim ens pot portar a fer moltes millores de fons, doncs solen ser senzilles i poden repercutir molt en el temps de resolució.

Tot i això, no ens centrarem gaire en les millores de fons, tot i que algunes tenen un impacte força important (i n'hi ha algunes bastant sorprenents, els canvis de màquina virtual de Java o inclús de plataforma tenen efectes importants sobre la velocitat); tot i que s'ha realitzat unes quantes millores (doncs agilitzen el procés de prova).

La mesura que farem servir serà poc rigorosa; ens fixarem en el nombre de nodes que hem hagut d'examinar per tal de trobar la solució, però donarem importància al temps de resolució. Efectivament, podríem tenir mètodes que donarien resolucions en molts pocs nodes, però que serien molt lentes (un exemple extrem seria calcular una heurística perfecta [és a dir, què ens digui exactament quants moviments necessita una posició per ser resolta], que seria extremadament lenta d'avaluar [bàsicament, es tractaria de posar un *solver* de Sokoban complet en el càlcul de l'heurística]), però que portaria directament a una solució.

Considerarem que una bona millora ens retalla el nombre de nodes significativament reduint alhora el temps de resolució.

3.4.1 Detecció simple de blocs encallats

La primera millora que vam incloure va ser una senzilla detecció de posicions sense solució; segons la classificació anterior, una millora específica del Sokoban. Podem marcar sobre el mapa posicions en les quals sota qualsevol condició una caixa mai no podrà arribar a un objectiu.

Això poda molt bé l'arbre, ja que col·ocar una caixa en aquestes posicions fa que una situació i totes les seves derivades siguin impossibles de resoldre.

Addicionalment, aquest càlcul és molt poc costós, el podem realitzar al principi de tot i fer-lo servir durant tot el procés de resolució.

Implementació

Al principi de l'execució de l'algorisme, creem un array de booleans on, per cada casella del mapa, hi posarem si una caixa en aquesta posició pot ser empesa fins una casella objectiu.

El criteri que fem servir és el següent:

- Una casella objectiu (inclús si té una caixa o l'home del magatzem a sobre), no està encallada.
- Una casella paret està encallada.
- Una casella a veïna d'una casella no encallada b no estarà encallada si podríem empènyer un bloc en a fins a b.

Fem servir un senzill algorisme iteratiu que va 'descobrint' pas a pas noves caselles no encallades, fins que en un pas no en troba cap.

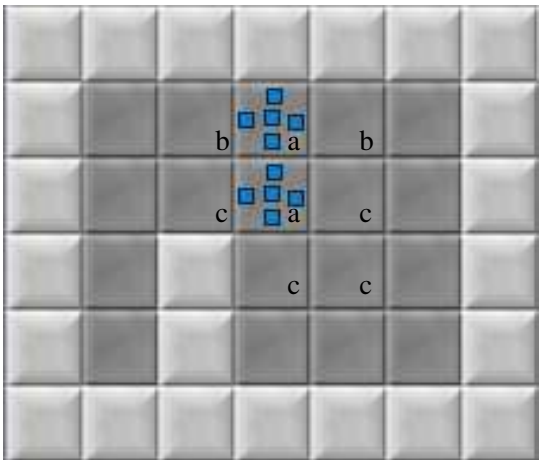

Figura 3.6: Exemple de caselles encallades

A la figura 3.6 veiem com quedaria el càlcul de blocs encallats en un mapa d'exemple. Les caselles etiquetades amb a no estan encallades, doncs són caselles objectiu. Les caselles amb b no estan tampoc encallades, doncs un bloc es podria empènyer a l'objectiu superior, així com les c, que porten a qualsevol dels dos objectius. Un bloc a la resta de caselles mai no podria arribar a una casella objectiu.

Avaluació

El cost de la detecció de blocs encallats és despreciable. Només necessitem executar-lo un cop i podem fer-lo servir en tots els passos de l'algorisme.

3.4.2 Refinament heurístic: distància en empentes

Una altre millora específica al Sokoban és intentar trobar una heurística més potent.

La nostra heurística inicial assigna un 'cost' a cada casella del mapa, en particular la distància de Manhattan fins a l'objectiu més proper. Ja hem vist que aquest cost és una cota inferior del nombre d'empentes necessàries per resoldre un mapa (i per tant, del nombre de moviments d'home, ja que en un moviment d'home només podem empènyer un bloc).

Podem refinar aquesta heurística aplicant un raonament similar al dels blocs encallats. Una casella objectiu té heurística 0, doncs no es necessita cap empenta per portar un bloc en aquesta casella cap a un objectiu. Les caselles des d'on podem portar un bloc a un objectiu amb una empenta hauran de tenir com valor d'heurística 1, les caselles des de les que podem empènyer un bloc a una d'aquestes caselles tindran valor 2 i així successivament.

És fàcil de veure que aquesta heurística també és admissible i igualment vàlida per `SituacioSimple` i `SituacioComplexa`.

Podem integrar aquest càlcul en el procés de detecció de blocs encallats sense afegir-hi un cost addicional[1]. Aquesta nova heurística resol dos problemes que tenia l'heurística anterior:

- Al fer servir la distància de Manhattan, l'heurística 'considerava' que el camí més curt entre un bloc i un objectiu és la línia recta, quan poden haver-hi obstacles pel mig. A la figura 3.7 veiem com la distància de Manhattan (representada per les fletxes negres) estima una distància de 2. La nova heurística (fletxes blanques) ens dóna una distància de 8.

- L'heurística no tenia en compte que molts moviments de blocs no són possibles donat que l'home no es pot col·locar en la posició necessària per empènyer-lo.

Addicionalment, la nova heurística és dominant respecte a la distància de Manhattan.

[1] De fet, en la implementació, no ens hem preocupat de les caselles encallades: se'ls hi assigna cost zero (quan realment hauria de ser cost infinit). L'heurística millorada s'ha de combinar forçosament amb la detecció de blocs encallats; en cas contrari dóna resultats dolents: veu favorable portar blocs cap a caselles encallades!

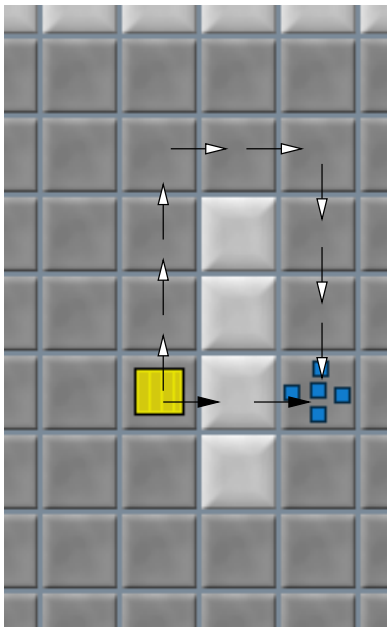

Figura 3.7: Exemple d'obstacles en l'heurística

3.4.3 Llista de nodes a analitzar

Com hem dit, tot i que ens vam centrar en millores específiques de Sokoban, vam implementar un parell d'optimitzacions genèriques i de fons.

La implementació de la llista de nodes a analitzar va passar per diverses fases en les que fèiem servir diferents estructures de dades. Les operacions que ha de suportar aquesta llista són reduïdes:

- Poder afegir nodes.
- Poder extreure (eliminant-l'ho) el node de menor cost (segons el mètode de cerca) de la llista.

De cara a aconseguir això, teníem dues opcions: mantenir la llista ordenada (fent que el cost de treure el menor de la llista fos mínim, però segurament incrementant el cost d'afegir nodes) o bé no mantenir-la (així treure el de menor cost possiblement fos lent).

La solució final va ser mantenir subllistes organitzades per costos, és a dir, tenir una llista per cada possible valor de cost.

Avaluació

Aquesta solució és força bona. Mantenim les subllistes en una estructura ordenada a la qual podem accedir en temps logarítmics (en arbre) i les subllistes tenen temps d'accés, afegir i eliminar gairebé constants (són conjunts hash).

3.4.4 Eliminar nodes redundants

Donat que mantenim una llista amb tots els nodes que hem analitzat, resulta fàcil afegir una comprovació simple: no afegir mai un node a la llista de nodes a analitzar si ja l'hem considerat en el passat. Això de vegades s'inclou en la definició de l'A*, però nosaltres vam estendre aquesta millora genèrica inclús si no fem un A*.

Avaluació

Donat que la cerca en la llista de nodes analitzats és força ràpida (fa servir taules hash), el cost és petit respecte a la multitud de posicions que ens estalviem.

3.4.5 Optimitzacions per profiling de heap

Tot i que ens volíem concentrar en millores algorísmiques més que en millores tècniques, vam decidir passar el programa pel profiler incorporat en la màquina virtual Java i mirar si podíem fer millores de fons.

El resultat va ser sorprenent. El profiling de l'espai heap revela que més del 70% de l'ús de memòria es feia en un punt que se'ns havia passat per alt. L'algorisme emmagatzema dins de cada node de l'arbre la llista de moviments per arribar a aquesta posició.

Ara bé, un cop arribem a certes profunditats de l'arbre, aquesta llista de moviments és força gran (centenars de moviments); si afegim que l'elecció d'estructura de dades no és massa òptima (un String; aproximadament ens costa 2 bytes per moviment), és fàcil de veure que estem desaprofitant la memòria. En resolucions en que arribem a 60.000 nodes, podem tenir perfectament una mitjana de 100 moviments per node, així estaríem fent servir $60.000 \cdot 100 \cdot 2 = 12.000.000$ bytes de memòria.

Vam decidir canviar lleugerament la representació. Enlloc de mantenir la llista de moviments completa, només emmagatzemem un apuntador a la posició "pare" del node i els moviments per arribar a aquesta posició des de la posició pare. Podem reconstruir la llista sencera de moviments d'una manera molt senzilla. Això comporta una reducció proporcional en ús de memòria a

la longitud mitjana de moviments de cada node; en el nostre cas hipotètic, reduiríem el consum de memòria 100 cops, baixant de 12 megabytes a 120 kilobytes.

Avaluació

El resultat és positiu. No obtenim una pèrdua de velocitat mesurable, però podem analitzar més nodes fent servir la mateixa quantitat de memòria. Tenint en compte que, a la pràctica, el nostre procés està limitat per la memòria, és força bo.

3.4.6 Optimitzacions per profiling de CPU

Els resultats de fer profiling de CPU van ser, per contra, molt menys interessants. No s'aprecien grans punts calents de l'aplicació. Els major punts calents a `SituacioComplexa`, per exemple, es troben en el procés de trobar els camins de cada situació, però tot i així, només representen típicament un 13% del temps de procés.

Donat que, com vam veure, el procés no està massa limitat per CPU, vam decidir no invertir temps en optimitzar per CPU, doncs els beneficis obtinguts possiblement no serien massa rendibles.

4. Desenvolupament del projecte

Partint dels objectius definits a 1.4, les funcionalitats que incloem són:

- Poder introduir un mapa (o escollir-ne un d'inclòs).
- Escollir i configurar un mètode de resolució.
- Executar el procés de resolució i visualitzar els resultats.

Addicionalment, posem els següents requeriments:

- Poder executar aquest procés tant interactivament (amb una interfície gràfica) com no interactivament (per línia de comandaments).
- Fer que el programa sigui executable sobre diverses plataformes sense complicacions.
- Que el programa sigui entenedor i fàcil d'estendre.

El primer pas de desenvolupament, doncs, és escollir el llenguatge i les llibreries que farem servir.

Els requeriments d'interfície gràfica **i** portabilitat ens redueixen molt les possibilitats. Les llibreries d'interfície "lliures" més esteses són possiblement:

GTK+ Popularitzada pel programa 'The Gimp' i implementada en C, disposa de *bindings* oficials per C++, Java, Python i Perl, a part de C, així com *bindings* extraoficials per Ruby, Lisp, Haskell i un munt de llenguatges més.

QT Proporciona un entorn complet de desenvolupament d'aplicacions més enllà de la seva llibreria per interfícies. Està molt orientada al C++, però hi han projectes per fer servir QT des d'altres llenguatges.

wxWindows També orientada a C++, en lloc d'implementar un conjunt de *widgets* propis, fa servir els *widgets* de la plataforma on s'executa (això teòricament proporciona un *look and feel* nadiu i més rendiment).

Swing És el component d'interfícies de la plataforma Java.

Ens vam decidir per Swing des de Java, doncs permet l'execució des del navegador, els programes són directament executables sense recompilar en les diferents plataformes suportades i, a part de la opinió personal, Java disposa de gestió automàtica de la memòria, fet que suposa un gran avantatge en programes d'aquest tipus.

4.1 Model de desenvolupament

El procés de desenvolupament s'ha centrat, òbviament, en la implementació de l'algorisme de resolució. La primera versió funcional del programa (amb interfície funcional i la primera iteració del *solver*) va suposar un cost relativament petit de desenvolupament. L'evolució posterior comprenia refinaments d'interfície i millores de l'algorisme de resolució.

La gran part del temps de desenvolupament es va dedicar a l'algorisme de resolució, i per tant els capítols de resolució i d'avaluació de rendiment ens ofereixen una visió de l'evolució del projecte, doncs aquest va ser bàsicament un cicle repetitiu d':

1. Execució de proves
2. Anàlisi de resultats
3. Disseny de millora

Amb el qual intentàvem resoldre cada cop més mapes en menys temps.

Al mateix temps, durant l'ús del programa anàvem millorant i polint el programa en conjunt.

4.2 Mòduls

Per explicar l'estructura i disseny del projecte farem una anàlisi horitzontal, distingint les següents parts o mòduls:

- Interfície gràfica (`soko.app`, `soko.ui`)
- Interfície de línia de comandaments (`soko.batch`)

- Serveis bàsics de Sokoban (`soko.base`)
- *Solvers* (`soko.solvers`)
- Serveis d'utilitat genèrics (`soko.util`)

Podem veure un esquema de la implementació a la figura 4.1.

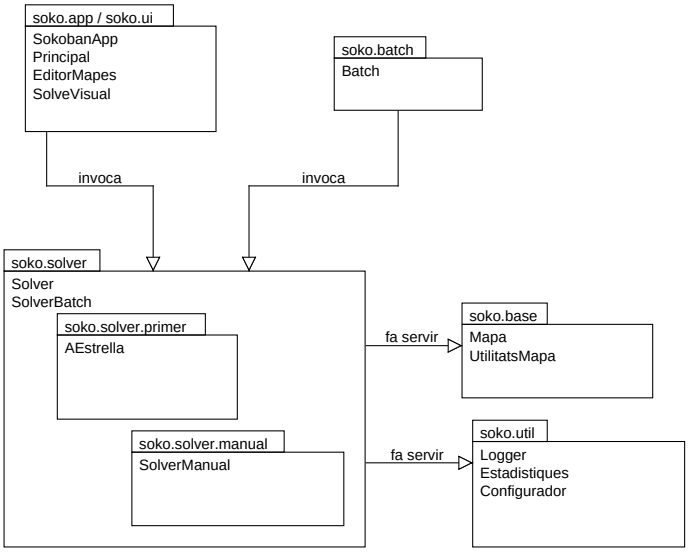

Figura 4.1: Estructura general

4.2.1 Interfície gràfica

Implementem una interfície gràfica en Swing que es pot executar com aplicació estàndard o com *applet*.

La seqüència d'ús habitual del programa serà:

1. Escollir mapa o l'editor de mapes.
2. Escollir el *solver*.
3. Dissenyar el mapa (si s'ha escollit l'editor).
4. Configurar el *solver*.
5. Executar resolució.

6. Visualitzar solució.

El procés parteix d'una finestra implementada a `soko.ui.Principal`. Podem veure el seu disseny a la figura 4.2.

Figura 4.2: La finestra inicial del programa

Conté codi per omplir els desplegables amb els mapes i *solvers* inclosos i llençar el *solver* escollit amb el mapa adient (o invocar l'editor de mapes). Això està implementat amb els serveis de configuració genèrics que detallem a la secció 4.2.5.

Editor de mapes

L'editor ens proporciona una graella de tamany configurable i una paleta amb les diferents possibilitats que podem posar a cada casella. El veiem a la figura 4.3.

La implementació està a `soko.ui.EditorMapes`. Fent clic a la graella s'omple la casella amb l'última icona que hem seleccionat a la paleta.

Visualitzador de solucions

Ens permet veure els moviments que conformen una solució, el podem veure a la figura 4.4.

De fet, la implementació de `soko.ui.SolveVisual` també s'encarrega d'iniciar el procés de resolució.

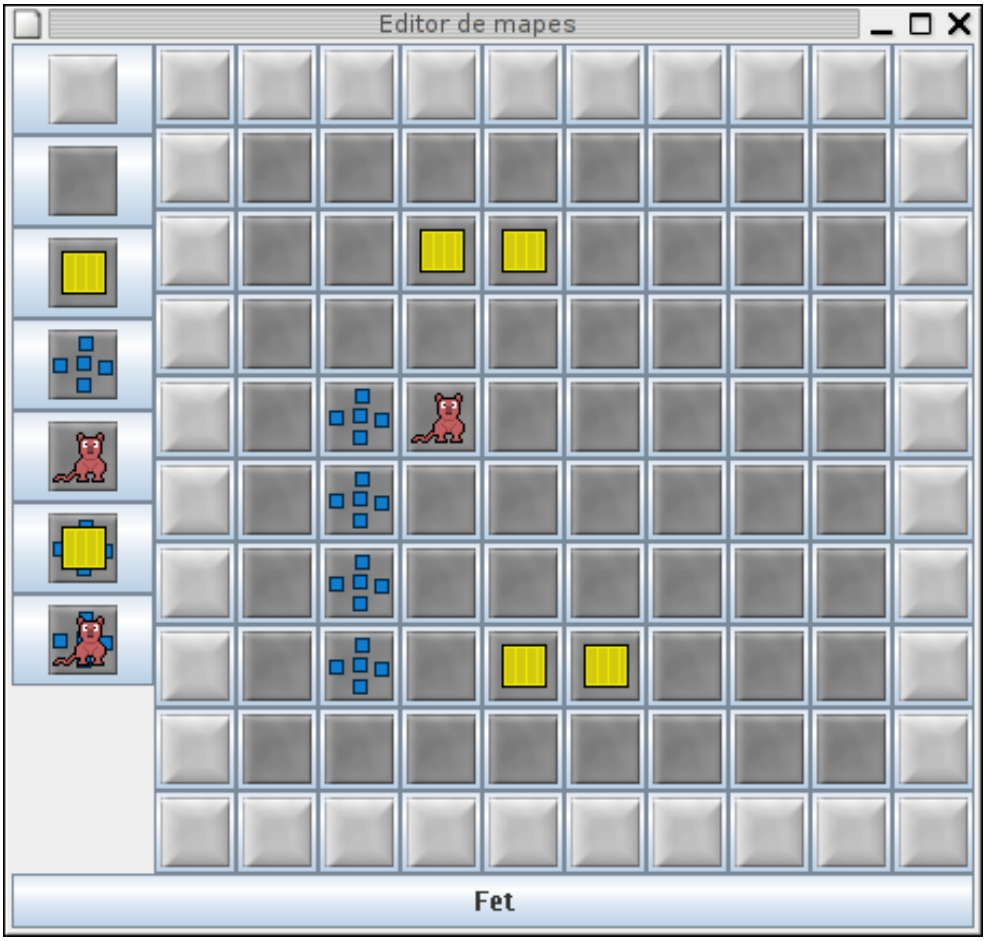

Figura 4.3: L'editor de mapes

4.2.2 Interfície de línia de comandaments

S'ocupa de realitzar el mateix procés que la interfície gràfica (excepte l'edició de mapes; només podem llençar el procés sobre els mapes empaquetats). Com en l'interfície gràfica, això es fa 'transparentment' mitjançant els serveis de configuració (veure 4.2.5).

Invocant el programa des de la línia de comandes, aquest agafa com arguments el mapa a resoldre (d'entre els inclosos amb el programa), el mètode de resolució i els paràmetres d'aquest. Un cop resolt, es mostra el resultat per pantalla.

Hem inclòs una referència de les opcions vàlides a l'apèndix B.

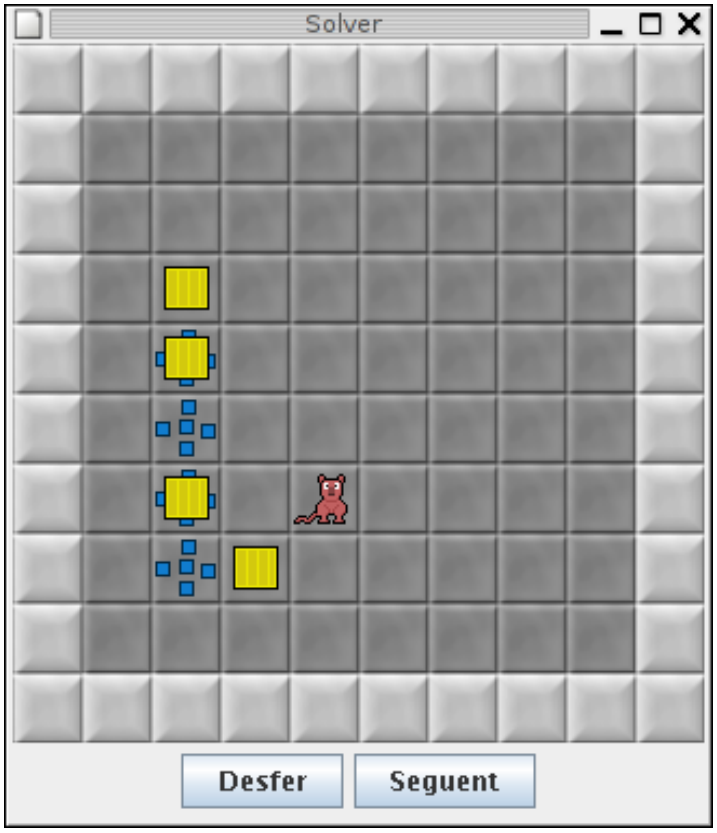

Figura 4.4: El visualitzador de solucions

4.2.3 Serveis bàsics de Sokoban

Al paquet `soko.base` incloem tot un seguit de funcions de base pel joc del Sokoban, que son:

- Representació de mapes (`soko.base.Mapa`, `soko.base.MapaComplet`)
- Funcions vàries sobre aquestes representacions (dins de la classe `soko-.base.UtilitatsMapa`):
  - Manipulació de mapes
  - Lectura de mapes des de fitxers
  - Càlculs de camins

La representació és un array bidimensional de caràcters, on cada possibilitat del mapa (casella lliure, objectiu, caixa, caixa sobre objectiu, etc.) està

codificada amb un caràcter. És raonablement compacta (malgrat que Java fa servir 2 bytes per caràcter, quan amb 3 bits en tindríem prou) i molt fàcil de cara a treballar amb ella.

4.2.4 Solvers

Per permetre l'ús de diferents implementacions de *solvers*, vam dissenyar una jerarquia de classes per a poder codificar diverses implementacions i fer-les servir des del programa.

Definim una interfície principal `soko.solvers.Solver` que representa un *solver*, és a dir, una entitat identificable que donat un mapa, ens va retornant els moviments per resoldre'l.

Aquesta interfície, no obstant, no es correspon amb la majoria de processos de resolució del Sokoban (però potser sí d'altres problemes). Un *solver* de Sokoban normalment calcularà la solució sencera inicialment. Per acomodar aquests algorismes, definim una classe auxiliar, `soko.solvers.SolverBatch`, en la qual podem definir un mètode que calculi la solució (a implementar) i aquesta s'encarrega de presentar aquesta solució pel mecanisme de `Solver`; és a dir, va entregant els moviments de la solució d'un en un.

Podem veure aquesta estructura en la figura 4.5.

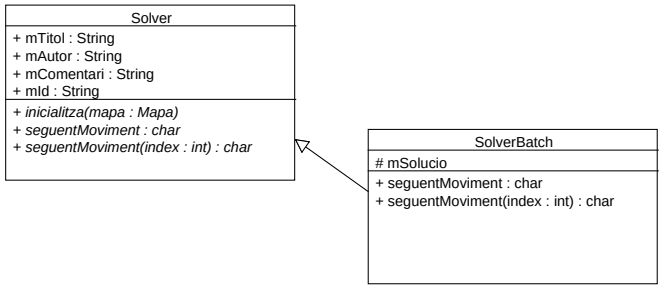

Figura 4.5: La jerarquia de Solvers

Aquesta estructura és pràctica, doncs sí que tenim un "`Solver`" que va donant els moviments d'un en un: el nostre mecanisme per poder experimentar amb els mapes 'manualment'. Així doncs, podem integrar un sistema per jugar al Sokoban en la nostra aplicació amb el teclat (això sí, per fer-lo mínimament decent vam haver de fer força concessions de disseny; es tracta d'un procés força diferent als *solvers* automàtics).

4.2.5 Serveis d'utilitat genèrics

El paquet `soko.util` proporciona unes quantes utilitats addicionals.

`soko.util.Estadistiques` conté una petita implementació numèrica per calcular el factor de ramificació efectiu del mapa (que veurem a 5.1.1), També, en conjunció amb la classe `soko.util.Operacio`, té un profiler rudimentari que ens permet extreure fàcilment informació de rendiment des de l'aplicació mateixa[1]. Aquesta informació es pot presentar per pantalla formatejada amb els mètodes de `soko.util.Formatejador`.

`soko.util.Logger` proporciona un senzill mecanisme de logging[2] que fem servir a l'aplicació.

Servei de configuració

Vam implementar un sistema de configuració raonablement versàtil per poder implementar tant una interfície de línia de comandaments, com una interfície gràfica.

Establim una estructura de tipus mapa amb parells 'opció, valor' que farem servir tant per escollir el mapa i *solver*, com per les opcions de cada *solver* en concret.

En arrancar el programa, processem els arguments de línia de comandaments i els emmagatzemem en la nostra configuració; si s'ha escollit el mode batch, llencem el *solver* automàticament. Ara bé, fem servir un procés 'mandrós' per llegir la configuració: només llegim el mapa de configuració quan necessitem un valor. El procés que llegeix la configuració, si no troba l'opció ja configurada, fa una petició per pantalla a l'usuari. El resultat de cara a l'usuari és una experiència normal: l'aplicació demana les opcions quan les necessita.

[1] No obstant, la màquina virtual de Java de Sun proporciona profiling molt més sofisticat i precís. Tot i això, la nostra implementació ens permet obtenir resultats útils des de dins de l'aplicació i realitzar algunes mesures d'alt nivell

[2] Java 1.4 i posteriors proporcionen un servei de logging més elaborat; però vam decidir no fer-lo servir per poder ser compatibles amb Java 1.3

5. Avaluació de rendiment

Dins del procés de treball sobre el problema ja vam veure que una bona anàlisi del rendiment no només seria valuosa al final del treball per il·lustrar les fites assolides, sinó que és indispensable per orientar-nos en les millores que realitzaríem en el programa.

5.1 Metodologia de l'avaluació

El procés d'avaluar el rendiment de qualsevol aplicació és força complex i subjectiu, per tant cal trobar bons criteris per avaluar el comportament del programa.

En cada execució del programa tindrem tres factors per avaluar l'efectivitat del *solver*:

- El mapa escollit.
- La solució obtinguda.
- Mesures que podem extreure del procés d'execució.

És difícil 'valorar' la dificultat d'un mapa. Podem pensar en criteris 'objectivament mesurables', com ara:

- Nombre mínim de moviments requerits per resoldre'l.
- Tamany del mapa.
- Nombre de caixes.

Però, tot i que sol haver-hi una correlació entre aquestes variables i l'efectivitat del nostre algorisme, podem trobar fàcilment exemples extrems on podem veure que aquestes variables no tenen cap sentit; un mapa arbitràriament gran amb només una caixa i un objectiu en punts oposats del mapa sense obstacles al mig és gran i necessita molts moviments per resoldre'l, però definitivament, no el podem qualificar de 'difícil'.

Així doncs, quedem limitats a criteris subjectius. Donat que aquests tampoc són massa útils (l'autor, per exemple, troba la majoria de mapes dels jocs de prova impossibles de resoldre), durant l'anàlisi de resultats no entrarem en valoracions sobre la dificultat dels mapes de prova.

La 'qualitat' de la solució obtinguda, d'altra banda, sí que la podem analitzar en molts casos. Analitzarem l'execució de diverses variants de l'algorisme, amb algunes variants que ens garanteixen trobar solucions òptimes. Així, podrem comparar les solucions obtingudes amb solucions òptimes. Això ens permet valorar la 'dificultat relativa' de trobar una solució comparada amb la dificultat de trobar una que sigui òptima. De vegades, però, els algorismes que donen solucions òptimes no funcionen, i no podrem valorar si les solucions trobades amb altres algorismes ho són.

5.1.1 Mesures sobre l'execució

Així doncs, cal analitzar el tercer factor, totes les variables que podem extreure de l'execució del programa. Això pot ser tan simple com veure si l'algorisme arriba a una solució o no, el temps que triga en arribar-hi o mesures més sofisticades.

Com vam veure a 4.2.5 el nostre programa genera un seguit d'estadístiques d'execució. També podem mesurar el temps d'execució del programa i inclús fer servir el profiler de la màquina virtual de Java per obtenir més dades analitzables.

Entre les variables que genera el nostre programa, tenim:

- Nombre de nodes d'arbre analitzats.
- Profunditat de l'arbre en la qual s'ha trobat la solució. Això serà equivalent al nombre de moviments de la solució si fem servir `SituacioSimple` o el nombre d'empentes de bloc amb `SituacioComplexa`.
- Factor de ramificació efectiu de l'arbre de resolució. El càlcul intenta aproximar numèricament el valor del factor de ramificació b dins l'equació $1 + b + b^2 + \ldots + b^d = n$, on d és la profunditat de l'arbre i n, el nombre de nodes d'arbre analitzats.

S'ha de dir que la majoria d'aquestes mesures no tenen cap valor si no s'analitzen en conjunt. Una millora heurística, per exemple, podria reduir el nombre de nodes necessaris per trobar una solució i altres variables, però ser tant costosa de càlcul com per a fer-la contraproduent.

Addicionalment, les mesures de temps són problemàtiques; no són massa precises (sobretot si intentem analitzar el temps d'execució de parts petites de l'algorisme) i es veuen molt influenciades per factors externs.

5.1.2 Joc de proves

Existeixen innumerables mapes de Sokoban, de dificultats molt variades i amb diferents problemes plantejats.

Vam decidir que els 90 mapes de l'XSokoban eren un objectiu massa ambiciós inicialment; els desenvolupadors de Rolling Stone només resolen una petita part després d'anys de recerca. En aplicar l'algorisme IDA*, un algorisme de matching de grafs bipartits i taules hash per reconèixer posicions repetides van veure que només podien resoldre 5 mapes, i per incrementar aquest nombre s'ha necessitat l'aplicació de tècniques molt sofisticades. Per posar un exemple de mapa d'XSokoban simple (estan més o menys ordenats per dificultat), a la figura 5.1 podem veure el tercer mapa d'XSokoban; té 11 caixes, i no ha estat resolt per humans en menys de 312 moviments i 134 empentes de bloc. Considerant un factor de ramificació d'1,05 (una estimació molt optimista si no fem servir unes quantes millores; considerem que és poc freqüent trobar-se en una posició on no hi hagin dos moviments possibles) si considerem moviments d'home, això ens dóna més de 4 milions de moviments a considerar per trobar una solució mitjançant una cerca d'amplada prioritària.

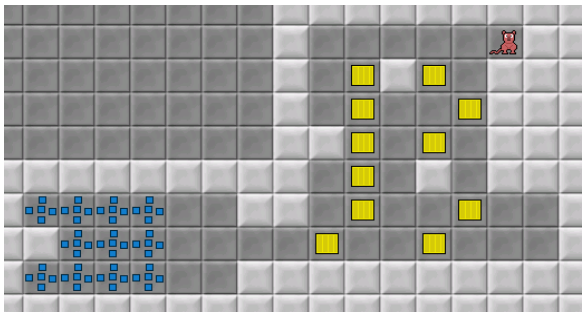

Figura 5.1: El tercer mapa d'XSokoban

Existeix un conjunt de 54 mapes creats per Yoshio Murase que són força

més simples però que tot i així són raonablement complicats pels éssers humans. Per exemple, el mapa de la figura 5.2 es pot resoldre en 90 moviments, examinant només unes 16000 posicions mitjançant amplada prioritària; i podem dir que no és precisament un mapa de resolució trivial.

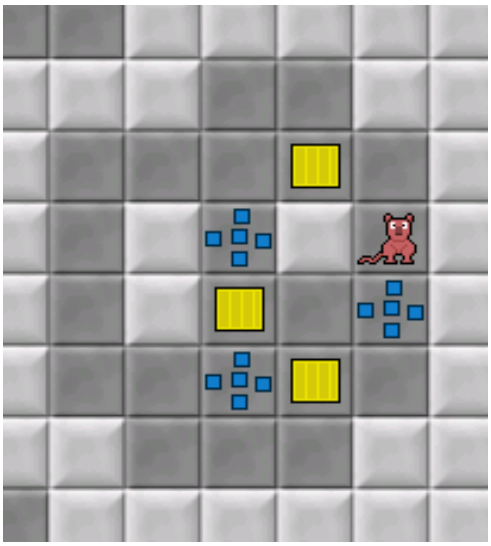

Figura 5.2: Exemple de mapa de Murase

El conjunt de Murase també ofereix mapes força complicats. Per fer-nos una idea, només hem resolt 35.

Per tal d'evidenciar certs problemes en el nostre programa, també farem servir mapes creats artificialment que ressalten les nostres limitacions; aquests seran sempre mapes de resolució òbvia per un humà.

5.1.3 Valoració de les solucions

Podem establir tres criteris diferents per avaluar l'èxit en la resolució d'un mapa:

- Trobar una solució.
- Trobar una solució òptima en quant a nombre de moviments d'home.
- Trobar una solució òptima en quant a nombre d'empentes de caixes.

Hi ha configuracions del nostre algorisme que garanteixen trobar solucions òptimes (amplada prioritària i A*) en un criteri, però (generalment), no són

Configuració				Resultat				
Situació	Enc.	Ordre	Heurística	CPU	♯ M.	♯ E.	♯ P.	Ram.
Simple	$\surd$	A*	Manhattan	∞	na	na	na	na
Complexa	$\times$	BFS	Empentes	12s	3	2	22	1,22

Taula 5.1: Exemple de resolució

òptimes amb els dos criteris alhora. La cerca *best-first* tampoc sol donar solucions òptimes.

El nostre objectiu, com veurem, serà trobar una solució, tot i que no sigui òptima.

Mostrarem els resultats d'una execució en una taula com la de 5.1, amb una fila per cada configuració del *solver* que provem:

- Configuració del *solver*
  - Tipus de situació: simple o complexa
  - Detecció de blocs encallats: $\surd$ o $\times$
  - Tipus de cerca: amplada prioritària, *best-first* o A*
  - Heurística: Manhattan o empentes
- Resolució
  - Temps de CPU per arribar a la solució
  - Nombre de moviments de la solució
  - Nombre d'empentes de la solució
  - Nombre de situacions analitzades per arribar a la solució

Indicarem amb una terna (∞, na, na, na) que el programa no ha aconseguit trobar una solució amb aquesta configuració (típicament, per falta de memòria a la màquina virtual[1]).

[1]Podríem ampliar els recursos, però vam decidir mantenir-los en el nivell per defecte, doncs estableix un llindar de resolució molt pràctic. Si no es troba la solució, el programa es queda sense memòria força de pressa. Així podem avaluar ràpidament.

5.2 Resultats

5.2.1 Mapes artificials

Petit

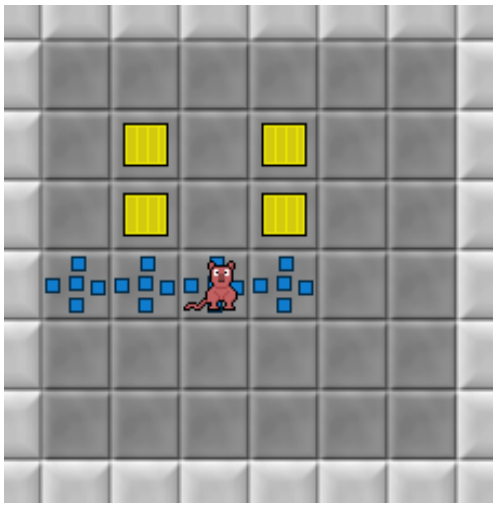

Figura 5.3: Artificial - Petit

El mapa 'Artificial - Petit' (5.3) és un típic mapa assequible de resolució; trivial de resoldre per un humà. No obstant, veiem que una cerca en amplada prioritària (sense coneixement del domini del problema) no és suficient per resoldre'l.

Afortunadament, qualsevol de les altres millores és suficient per trobar una solució. Dins l'àmbit de `SituacioSimple`, veiem que de cara a trobar una solució òptima, l'heurística de Manhattan (amb detecció de blocs encallats) funciona més be que la suposadament millor heurística d'empentes. En canvi, quan eliminem la restricció d'optimalitat, l'heurística d'empentes és lleugerament millor (però la solució que ens proporciona és pitjor).

En canvi, amb `SituacioComplexa` l'heurística d'empentes és inequívocament més bona. Amb la cerca *best-first*, de fet, ens porta **directament** a una solució.

En aquest mapa ens adonem del que serà la tònica general, les restriccions d'optimalitat en les solucions (tant de moviments amb `SituacioSimple` com d'empentes amb `SituacioComplexa`) són el factor més important de cara a determinar la velocitat amb que trobarem una solució. Si ens con-

formem amb solucions no òptimes obtenim millores de com a mínim 60x en `SituacioSimple` i 3x en `SituacioComplexa`.

Fàcil

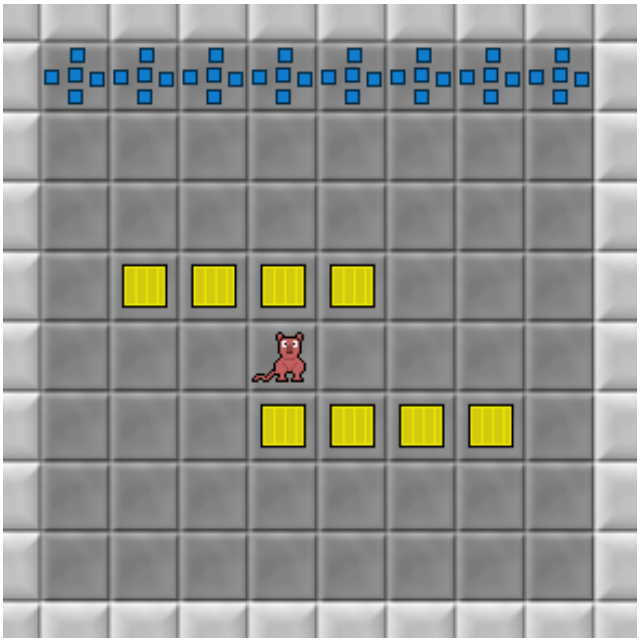

Figura 5.4: Artificial - Fàcil

El mapa 'Artificial - Fàcil' (5.4) ens mostra com un problema aparentment molt senzill necessita aplicar unes quantes tècniques abans que es pugui resoldre satisfactòriament.

L'autor, sense trencar-se massa les banyes, troba una solució en 97 moviments i 41 empentes.

Però veiem que en aquest mapa, si exigim optimalitat, el nostre algorisme no ens aconsegueix donar cap solució. Encara més, quan retirem el requisit, les solucions que trobem s'allunyen molt del que un humà aconsegueix sense dificultat (bé, en quant a nombre d'empentes, creiem que el mínim seria 38, que no està tant lluny del que obté l'algorisme).

Un altre fenomen que observem és que l'heurística d'empentes torna a ser inferior en certes condicions, aquí amb `SituacioComplexa`.

Un dels problemes que s'evidencien en aquest mapa són les interaccions entre caixes. És força fàcil que l'algorisme agrupi caixes accidentalment (sobretot gràcies a les heurístiques que, com hem vist, fan força atractiu portar

un seguit de caixes cap el mateix objectiu; una mena d'embús), de manera que quedin bloquejades i immòbils.

En canvi, veiem que la detecció de blocs encallats no representa cap millora. Sembla obvi, doncs l'única manera d'encallar un bloc (independentment dels altres) és portar-lo a les caselles inferiors del mapa; moviments que van totalment en contra de l'heurística. Tècnicament, portar més d'un bloc a les caselles laterals també els encalla (només hi ha una casella objectiu a cada marge, així que si hi ha més d'un bloc en un marge, només un podrà arribar a un objectiu), però això tampoc té massa sentit 'heurístic', ni tampoc es detectat per l'algorisme.

Cantonades

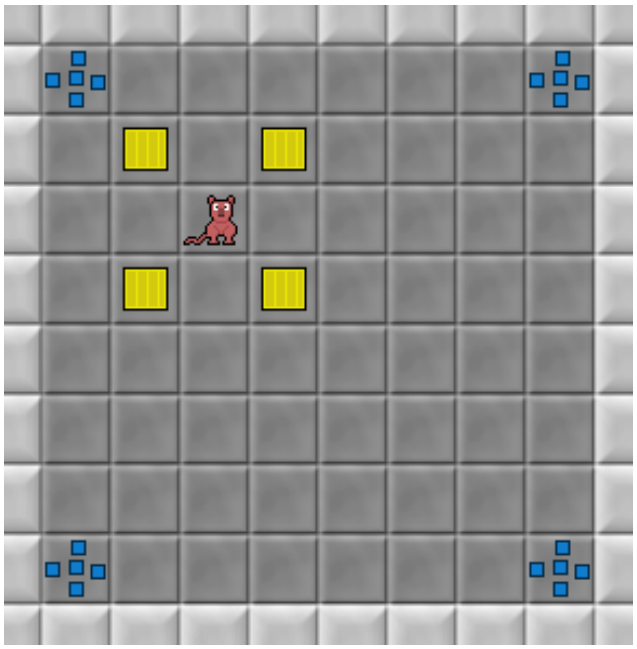

Figura 5.5: Artificial - Cantonades

Un altre cop, un mapa aparentment trivial resulta difícil de resoldre per l'ordinador. L'autor el resol en 46 moviments sense trencar-se les banyes.

El principal problema (a part que sense una cerca heurística la solució està a massa profunditat) és que amb les heurístiques implementades, tenim un greu problema: moure tots els blocs cap a la cantonada comporta un descens d'heurística, però no ens porta cap a una solució. Ho podem comprovar veient

que si variem la disposició dels blocs de manera que cada bloc comenci tenint un objectiu més proper diferent, el mapa es resol molt més fàcilment.

Aquest problema es veu compensat quan fem una cerca totalment heurística sense demanar una solució òptima. Llavors baixa suficientment el nombre de nodes a analitzar com per a trobar una solució.

Grup

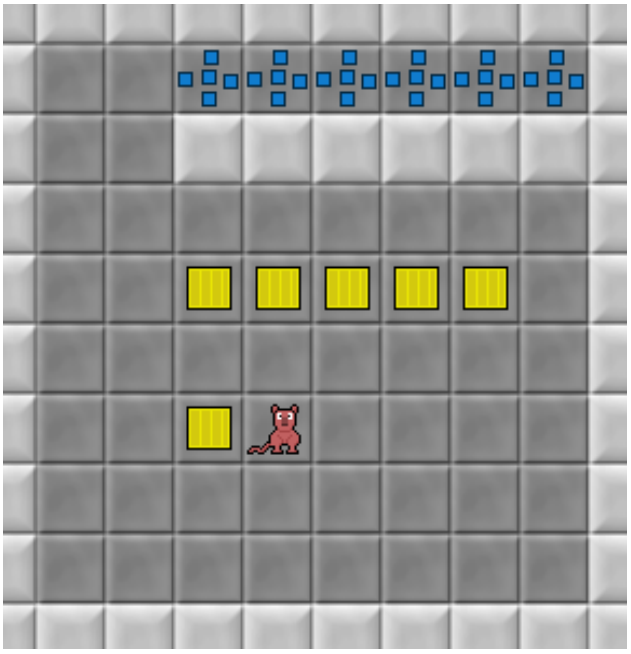

Figura 5.6: Artificial - Grup

Un altre problema que resulta intuïtivament simple per un humà és el dels grups de caselles objectius. En el mapa Artificial - Grup, veiem clar que la primera caixa l'hem de posar a la casella objectiu més 'profunda', la segona a la següent i així successivament.

Ara bé, amb la nostra heurística, un cop la primera caixa es posa sobre una casella objectiu, no hi ha cap 'interès heurístic' en moure-la cap als objectius més amagats. Això és un gran problema, doncs es creen 'embussos' de caixes. Mentre la cerca no es capaç de comprovar que no hi ha cap solució posant la primera caixa a l'objectiu més extern, no provarà cap combinació més. Un cop ho comprovi, només la portarà al segon objectiu. Això provoca una explosió combinatòria greu; de totes les possibles combinacions caixa-objectiu, la combinació bona la provarà l'última (si hi arriba).

Només amb aquest petit detall, la majoria de problemes d'XSokoban ja no poden ser resolts, doncs la majoria tenen grups de caselles objectiu similars. Aquest mapa d'exemple, tot i ser relativament simple (un humà troba la solució directament, l'autor el resol en 151 moviments), no es pot resoldre amb la nostra aplicació, ni tan sols aplicant totes les millores i renunciant a l'optimalitat.

5.2.2 Mapes Murase

Murase 2

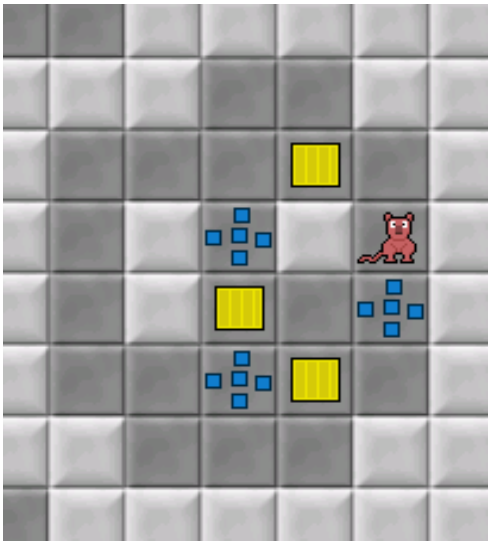

Figura 5.7: Murase 2

El mapa Murase 2 és el primer amb el qual vam treballar (fora de mapes trivials per provar). Resulta molt interessant per l'autor, doncs aquest és incapaç de resoldre'l (tot i que ha contemplat la solució més d'un cop). Fent proves informals amb diversos subjectes humans, només n'hem trobat 1 que el sap resoldre.

Això és força curiós, doncs aquest mapa es resol relativament sense problemes amb una senzilla cerca en amplada prioritària sense afegir cap tècnica específica del Sokoban. Això està influenciat per ser un mapa força 'tancat' (sobretot amb `SituacioSimple`, els mapes oberts tenen la penalització que s'ha de trobar -a sobre dels moviments de blocs- els camins a les caixes) i amb un nombre petit de caixes; la interacció entre caixes sembla ser el major problema de l'algorisme.

Donat això, Murase 2 sembla un bon mapa per poder comparar totes les tècniques que hem fet servir.

Un altre cop, la cerca *best-first* aconsegueix els millors resultats, però amb un marge més estret que en altres casos; trobar una solució òptima no sembla ser el major problema (de fet, les solucions obtingudes amb *best-first* no s'allunyen massa de les solucions òptimes). Veiem que en un mapa tancat els blocs encallats són el major problema, aplicar la detecció de blocs encallats suposa la millora més gran de les tècniques aplicades (recordem que l'heurística d'empentes inclou la detecció de blocs encallats). Això ho veiem clarament al mapa, doncs només la meitat de les (11) caselles del mapa són 'no-encallades' de totes les caselles accessibles (22) del mapa. La poda de caselles encallades és, doncs, molt efectiva.

També podem comparar clarament la influència d'escollir `SituacioSimple` vers `SituacioComplexa`. Veiem que si escollim `SituacioComplexa`, la profunditat a la qual hem de descendir per trobar la solució baixa molt (de 90–124 nodes a 17–23) i els temps de resolució també són millors. No obstant (i gairebé anecdòticament), els nodes analitzats per unitat de temps són pitjors; no és sorprenent, doncs els càlculs de `SituacioComplexa` són més complicats i costosos. Però això es veu compensat per la reducció en el nombre de nodes que comporta.

Un altre resultat fàcil d'endevinar és que el factor de ramificació efectiu és més alt per `SituacioComplexa`; en el cas mig, hi ha més possibilitats d'empentes de bloc (fins a 4 per caixa en joc) que de moviments d'home (màxim 4). En tot cas, està lluny del que hem pogut arribar a veure en mapes artificials, factors de ramificació superiors a 3.

També comprovem que l'heurística (en aquest cas) no serveix de massa ajuda a l'hora de trobar una solució òptima. Sense la detecció de blocs encallats amb `SituacioSimple` l'únic que aconseguim és arribar a l'últim moviment més aviat. Fent servir `SituacioComplexa` s'aconsegueix una millora major, però encara relativament petita. Això no vol dir que l'heurística sigui inútil; com veiem, si deixem que l'heurística guiï completament la cerca, sí que s'aconsegueix un resultat més ràpid (però la solució obtinguda no és òptima).

Murase 9

Murase 9 és un dels mapes més complicats que resol l'algorisme. Amb una solució mínima de 22 empentes de bloc (i no sabem el mínim de moviments per resoldre'l) i 6 caixes, hem de fer servir `SituacioComplexa`, cerca heurística i detecció de blocs encallats per poder trobar alguna solució.

Un altre cop, veiem que la distància de Manhattan torna a ser més efec-

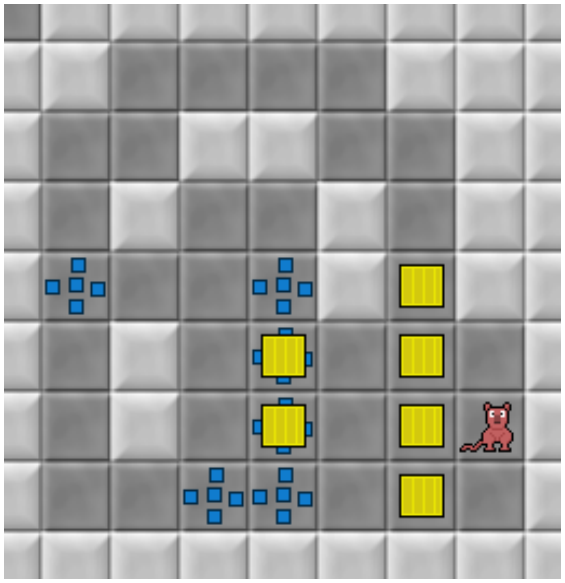

Figura 5.8: Murase 9

tiva; no només ens porta a les solucions més ràpidament, sinó que ens porta (fent una cerca *best-first*) cap a una solució força millor (en canvi, la solució d'A* amb l'heurística d'empentes és, anecdòticament, lleugerament millor en quant a nombre de moviments d'home); de fet, amb *best-first* la distància de Manhattan és molt superior: ens porta a la solució examinant un 66% dels nodes que necessita l'heurística d'empentes, inclús ens porta a una solució gairebé òptima (només dues empentes més) en pràcticament la meitat de nodes i temps.

Això és molt sorprenent. Si comparem els valors heurístics que generem, només hi ha quatre caselles (no encallades) en les quals obtenim valors diferents; i en aquestes, la diferència és només d'1 punt. Sembla ser, doncs, que el fet que una heurística sigui dominant, no és necessàriament millor.

5.2.3 Resolució global dels Mapes de Murase

A la figura 5.2 tenim una vista 'global' de l'efectivitat de les diferents opcions de l'algorisme de cara a resoldre els mapes de Murase. Cada línia ens diu el nombre de mapes resolts de Murase en una configuració, dels 54 mapes que existeixen.

Com veiem, potser la millora més important és la detecció de blocs encallats. Des de la substancial millora que representa comparat amb el mecanisme de resolució més bàsic (`SituacioSimple` en amplada prioritària), on

triplica el nombre de mapes resolts, podem veure que sempre proporciona una millora notable.

Cal dir, però, que els mapes de Murase són particularment petits; així doncs, una part important de les caselles seran encallades i la importància de la detecció serà més gran, teòricament, que en un mapa gran.

Aplicar l'A* i heurístiques, per contra, no representa un canvi molt gran, proporcionant-nos potser una millora d'un o dos mapes més resolts respecte la configuració equivalent en amplada prioritària.

Aplicar la cerca *best-first* proporciona millores respectables respecte l'A* en certes configuracions (i s'estableix com l'algorisme que resol més mapes), però això té com a cost renunciar a obtenir solucions òptimes en la majoria dels cassos.

Finalment, cal ressaltar la similitud entre l'efectivitat de les dues heurístiques aplicades. No hi han grans diferències d'efectivitat, i cap de les dues és millor en totes les configuracions.

Configuració				Resolts
Situació	Enc.	Ordre	Heurística	
Simple	×	A. P.	Manhattan	7
Simple	√	A. P.	Manhattan	22
Simple	×	A*	Manhattan	8
Simple	√	A*	Manhattan	22
Simple	√	A*	Empentes	22
Simple	×	B. F. S.	Manhattan	19
Simple	√	B. F. S.	Manhattan	28
Simple	√	B. F. S.	Empentes	26
Complexa	×	A. P.	Manhattan	13
Complexa	√	A. P.	Manhattan	29
Complexa	×	A*	Manhattan	15
Complexa	√	A*	Manhattan	31
Complexa	√	A*	Empentes	31
Complexa	×	B. F. S.	Manhattan	22
Complexa	√	B. F. S.	Manhattan	33
Complexa	√	B. F. S.	Empentes	35

Taula 5.2: Mapes resolts per configuració

5.2.4 Taules

De cara a facilitar la llegibilitat, incloem en aquesta secció les taules de resultats obtingudes en la resolució dels mapes analitzats.

Configuració				Resultat				
Situació	Enc.	Ordre	Heurística	CPU	♯ M.	♯ E.	♯ P.	Ram.
Simple	×	A. P.	Manhattan	∞	na	na	na	na
Simple	√	A. P.	Manhattan	6,85	19	8	46147	1,68
Simple	×	A*	Manhattan	2,31	19	8	12116	1,55
Simple	√	A*	Manhattan	1,24	19	8	6766	1,50
Simple	√	A*	Empentes	1,33	19	8	7620	1,51
Simple	×	B. F. S.	Manhattan	0,06	27	8	115	1,09
Simple	√	B. F. S.	Manhattan	0,05	27	8	115	1,09
Simple	√	B. F. S.	Empentes	0,04	31	10	107	1,07
Complexa	×	A. P.	Manhattan	∞	na	na	na	na
Complexa	√	A. P.	Manhattan	6,36	33	8	13298	3,12
Complexa	×	A*	Manhattan	0,14	29	8	114	1,60
Complexa	√	A*	Manhattan	0,14	29	8	114	1,60
Complexa	√	A*	Empentes	0,09	29	8	82	1,52
Complexa	×	B. F. S.	Manhattan	0,08	73	20	27	1,02
Complexa	√	B. F. S.	Manhattan	0,07	73	20	26	1,02
Complexa	√	B. F. S.	Empentes	0,03	35	10	11	1,00

Taula 5.3: Resolució d'Artificial Petit

Configuració				Resultat				
Situació	Enc.	Ordre	Heurística	CPU	♯ M.	♯ E.	♯ P.	Ram.
Simple	×	A. P.	Manhattan	∞	na	na	na	na
Simple	√	A. P.	Manhattan	∞	na	na	na	na
Simple	×	A*	Manhattan	∞	na	na	na	na
Simple	√	A*	Manhattan	∞	na	na	na	na
Simple	√	A*	Empentes	∞	na	na	na	na
Simple	×	B. F. S.	Manhattan	0,36	160	46	1080	1,02
Simple	√	B. F. S.	Manhattan	0,35	160	46	1080	1,02
Simple	√	B. F. S.	Empentes	0,28	128	44	875	1,02
Complexa	×	A. P.	Manhattan	∞	na	na	na	na
Complexa	√	A. P.	Manhattan	∞	na	na	na	na
Complexa	×	A*	Manhattan	∞	na	na	na	na
Complexa	√	A*	Manhattan	∞	na	na	na	na
Complexa	√	A*	Empentes	∞	na	na	na	na
Complexa	×	B. F. S.	Manhattan	0,13	156	40	50	1,01
Complexa	√	B. F. S.	Manhattan	0,13	156	40	50	1,01
Complexa	√	B. F. S.	Empentes	0,09	173	40	51	1,01

Taula 5.4: Resolució d'Artificial Fàcil

Configuració				Resultat				
Situació	Enc.	Ordre	Heurística	CPU	♯ M.	♯ E.	♯ P.	Ram.
Simple	×	A. P.	Manhattan	∞	na	na	na	na
Simple	√	A. P.	Manhattan	∞	na	na	na	na
Simple	×	A*	Manhattan	∞	na	na	na	na
Simple	√	A*	Manhattan	∞	na	na	na	na
Simple	√	A*	Empentes	∞	na	na	na	na
Simple	×	B. F. S.	Manhattan	1,34	142	32	6534	1,04
Simple	√	B. F. S.	Manhattan	1,33	142	32	6534	1,04
Simple	√	B. F. S.	Empentes	1,99	79	28	10259	1,09
Complexa	×	A. P.	Manhattan	∞	na	na	na	na
Complexa	√	A. P.	Manhattan	∞	na	na	na	na
Complexa	×	A*	Manhattan	∞	na	na	na	na
Complexa	√	A*	Manhattan	∞	na	na	na	na
Complexa	√	A*	Empentes	∞	na	na	na	na
Complexa	×	B. F. S.	Manhattan	3,52	87	28	8355	1,31
Complexa	√	B. F. S.	Manhattan	3,48	87	28	8355	1,31
Complexa	√	B. F. S.	Empentes	4,47	87	32	13217	1,28

Taula 5.5: Resolució d'Artificial Cantonades

Configuració				Resultat				
Situació	Enc.	Ordre	Heurística	CPU	♯ M.	♯ E.	♯ P.	Ram.
Simple	×	A. P.	Manhattan	1,69	90	17	16257	1,08
Simple	√	A. P.	Manhattan	0,38	90	17	2408	1,06
Simple	×	A*	Manhattan	1,90	90	17	16243	1,08
Simple	√	A*	Manhattan	0,40	90	17	2406	1,06
Simple	√	A*	Empentes	0,40	90	17	2407	1,06
Simple	×	B. F. S.	Manhattan	0,39	112	19	2155	1,04
Simple	√	B. F. S.	Manhattan	0,21	124	19	944	1,03
Simple	√	B. F. S.	Empentes	0,28	108	21	1585	1,04
Complexa	×	A. P.	Manhattan	0,62	90	17	2663	1,49
Complexa	√	A. P.	Manhattan	0,13	116	17	357	1,30
Complexa	×	A*	Manhattan	0,62	112	17	2426	1,48
Complexa	√	A*	Manhattan	0,14	116	17	346	1,29
Complexa	√	A*	Empentes	0,14	116	17	351	1,29
Complexa	×	B. F. S.	Manhattan	0,16	118	19	316	1,24
Complexa	√	B. F. S.	Manhattan	0,10	120	21	168	1,16
Complexa	√	B. F. S.	Empentes	0,11	126	23	228	1,16

Taula 5.6: Resolució de Murase 2

Configuració				Resultat				
Situació	Enc.	Ordre	Heurística	CPU	♯ M.	♯ E.	♯ P.	Ram.
Simple	×	A. P.	Manhattan	∞	na	na	na	na
Simple	√	A. P.	Manhattan	∞	na	na	na	na
Simple	×	A*	Manhattan	∞	na	na	na	na
Simple	√	A*	Manhattan	∞	na	na	na	na
Simple	√	A*	Empentes	∞	na	na	na	na
Simple	×	B. F. S.	Manhattan	∞	na	na	na	na
Simple	√	B. F. S.	Manhattan	∞	na	na	na	na
Simple	√	B. F. S.	Empentes	∞	na	na	na	na
Complexa	×	A. P.	Manhattan	∞	na	na	na	na
Complexa	√	A. P.	Manhattan	∞	na	na	na	na
Complexa	×	A*	Manhattan	∞	na	na	na	na
Complexa	√	A*	Manhattan	24,45	148	22	59914	1,57
Complexa	√	A*	Empentes	24,39	144	22	60167	1,58
Complexa	×	B. F. S.	Manhattan	∞	na	na	na	na
Complexa	√	B. F. S.	Manhattan	13,75	164	24	34427	1,47
Complexa	√	B. F. S.	Empentes	21,27	348	40	50807	1,26

Taula 5.7: Resolució de Murase 9

6. Conclusions

El joc del Sokoban ha estat una experiència molt interessant com a problema d'intel·ligència artificial. Mitjançant un procés iteratiu de millores de l'algorisme hem arribat a resoldre 35 dels mapes de Murase. Només considerant millores purament algorísmiques, hem de ressaltar que sense aplicar les tècniques heurístiques i les millores específiques, només n'arribaríem a resoldre 7.

A més, hem vist que el joc del Sokoban presenta certes particularitats interessants que s'han de tenir en compte a l'hora d'intentar resoldre'l algorísmicament.

6.1 Particularitats del Sokoban

Des del principi vam veure que hi ha una gran diferència entre la dificultat dels mapes per l'ordinador i per un humà. Mapes molt simples per l'ordinador són difícils de resoldre pels humans i a l'inrevés.

Els problemes fàcils pels humans que resulten difícils de resoldre per l'algorisme són especialment frustrants: molts dels problemes que presenten no tenen una solució 'neta' i clara d'implementar. Intuïtivament, creiem que la solució passa per prendre una visió de més 'alt nivell'; potser una evolució similar al pas de `SituacioSimple` a `SituacioComplexa` o l'ús d'alguna tècnica de resolució de problemes diferent (com ara tècniques de planificació).

6.1.1 L'heurística

Sobre el nostre algorisme, entrant en més detall, cal analitzar el valor de les funcions heurístiques en el procés de resolució.

Hem vist que és difícil trobar una heurística admissible per l'A* que sigui raonablement útil de cara a obtenir una solució òptima. Això ens indica que o bé l'A* no és el mètode més adient, o bé que trobar la solució òptima d'un mapa és un problema força més dur que trobar una solució al problema (de fet, per analitzar la viabilitat de la resolució dels mapes fem servir la cerca *best-first*, doncs sol ser considerablement més ràpida que l'A*).

La limitació d'haver de demostrar l'admissibilitat (si això fos possible) és una limitació; resulta difícil incorporar nous criteris que millorin l'heurística que s'ajustin a aquesta condició.

En tot cas, tenint en compte els costos de les solucions dels mapes, veiem que en un A* el cost de cada node és molt gran comparat amb els valors heurístics. Això fa que la influència de l'heurística sigui molt petita. En aplicar *best-first*, en canvi, aquest problema desapareix i l'heurística té més influència en la cerca, de manera que és més útil.

D'altra banda, veient la resolució d'alguns mapes resulta obvi que la majoria de mapes necessiten per la seva la resolució de moviments que són molt negatius des del punt de vista de qualsevol heurística (moviments que semblen inútils o contraproduents resulten vitals per la solució del problema).

Un altre punt interessant és que en alguns casos, l'heurística de Manhattan dóna millors resultats que l'heurística de distància en empentes de bloc (que és dominant).

L'altra millora heurística (que ja consideren els autors de Rolling Stone) és reflectir el fet que cada bloc ha d'anar a un objectiu diferent; les nostres heurístiques 'pensen' que més d'un bloc pot anar a la mateixa casella.

6.1.2 Grups d'objectius

A 5.2.1 vam veure el problema que plantegen un grup de caselles objectiu, un fenomen molt freqüent als mapes de Sokoban.

Com resoldre aquest problema?

Creiem que dins del context d'una cerca A* bàsica és molt complicat. No podem fer una poda, doncs certament no hi han nodes per podar. També és difícil aconseguir una heurística que reflecteixi això si volem mantenir l'admissibilitat.

Si som menys estrictes, sembla ser que sí podria ser possible. El camí de l'heurística sembla viable, però també planteja problemes. Si fem que el fet de posar la caixa en l'objectiu inapropiat sigui 'poc atractiu', potser evitarem que el programa passi per aquesta posició (i normalment, és necessari passar per caselles inapropiades per arribar a la bona).

Una solució (que implementa Rolling Stone) és que un cop una caixa entra en la zona de caselles objectius, afegir un moviment que la porta directament

a la casella bona i podar la resta. Això trenca una mica l'esquema de cerca però, aparentment, és molt efectiu.

6.2 Possibles millores

Aquest projecte vol ser també un punt de partida per treballar, no només en el domini del problema del Sokoban. L'aplicació ens pot servir per experimentar tècniques específiques del Sokoban, però també podem aplicar tècniques genèriques d'IA.

6.2.1 Estructura general

Podem afegir mecanismes de recerca en arbres (o d'altres tipus) que facin servir la funcionalitat específica de Sokoban (possiblement necessitant modificacions), permetent la comparació de diferents tècniques.

Tot i que l'aplicació s'ha dissenyat molt orientada al puzle del Sokoban, no seria excessivament costós dissociar la part del Sokoban i generalitzar l'aplicació per permetre especificar problemes diferents. De fet, el *solver* implementat ja separa clarament la part específica del Sokoban.

La funcionalitat es podria repartir en tres blocs; interfície, algorismes genèrics i problemes específics, i faria que la aplicació resultés molt útil com 'llibreria d'experimentació de problemes'.

6.2.2 Millores específiques del problema

A part dels punts que hem comentat sobre el procés de resolució, la implementació de l'algorisme encara pot rebre millores que podrien fer que es resolessin més mapes.

Donat que el programa encara està limitat pel seu consum de memòria, podríem aconseguir millors resultats optimitzant l'ús de memòria. Actualment, la llista de nodes analitzats és l'estructura de dades que ocupa més memòria. Encara es podria reduir més el seu tamany. Només la fem servir per veure si els nodes nous que analitzem ja hi són, així que només necessitem emmagatzemar les dades necessàries per fer aquestes comparacions. En la implementació actual, s'emmagatzema el node amb totes les dades (llista de camins, caselles encallades, ...) i això comporta un ús poc òptim de la memòria.

Més enllà, també podríem estalviar memòria fent servir estructures de dades més compactes. Emmagatzemem moltes dades en caràcters Java, una elecció molt pràctica de cara a facilitar la programació, però poc eficient.

També, de cara a poder analitzar els diferents factors algorísmics que afecten al procés de resolució, seria interessant una reestructuració lògica del programa que permetés trencar en unitats més petites les diferents parts configurables del procés del resolució per poder aïllar amb més precisió els factors que influeixen en la resolució.

6.3 Conclusions sobre la plataforma

En retrospectiva, sembla que l'elecció de Java i Swing ha estat encertada. Tot i que el plugin de Java no està massa estès, sí que hem tingut força èxit en l'execució del programa sobre diferents sistemes.

La implementació de la interfície va ser força directe i senzilla, i el resultat prou satisfactori. En quant al disseny de l'aplicació, l'orientació a objectes ha estat útil, tot i que certament, no imprescindible. La gestió automàtica de memòria ha resultat útil, però hauria pogut ser una bona idea disposar d'un major control sobre l'ús de la memòria. Aquest ha resultat un dels punts claus de l'efectivitat del procés de resolució i potser amb un ús més precís (tant des del punt de vista algorísmic com tècnic), podríem haver aconseguit millors resultats. Val a dir, però, que aquests problemes venen en part perquè hem escollit el camí de la simplicitat en la implementació, que també té els seus avantatges.

Bibliografia

[1] J. Culberson. Sokoban is PSPACE-complete. Technical report, Departament of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1997.

[2] Fukuda and Matsui. Finding all minimum-cost perfect matchings in bipartite graphs. *NETWORKS: Networks: An International Journal*, 22, 1992.

[3] Andreas Jughanns i Jonathan Schaeffer. Sokoban: Improving the search with relevance cuts. Es publicarà a: Journal of Theoretical Computing Science, 1999.

[4] Andreas Junghanns i Jonathan Schaeffer. Sokoban: A Challenging Single-Agent Search Problem. In *Proceedings IJCAI-97*, Nagoya, Japó, Agost 1997. Al taller Using Games as an Experimental Testbed for AI Research.

[5] Andreas Junghanns i Jonathan Schaeffer. Relevance Cuts: Localizing the Search. In *Proceedings CG-98*, Tsukuba, Japó, 1998.

[6] Andreas Junghanns i Jonathan Schaeffer. Single-Agent Search in the Presence of Deadlock. In *Proceedings AAAI-98*, pages 419–424, Madison/WI, USA, Juliol 1998.

[7] Andreas Junghanns i Jonathan Schaeffer. Domain-Dependent Single-Agent Search Enhancements. In *Proceedings IJCAI-99*, pages 570–575, Stockholm, Suècia, 1999.

A. Materials

A.1 Composició de la memòria

Aquesta memòria ha estat escrita fent servir els editors gVIM, joe, gedit i l'editor de text d'Eclipse, i processada mitjançant LaTeX 2$_\varepsilon$ (amb la implementació per Linux teTeX). Els gràfics s'han creat a partir de captures de pantalla de l'*applet* (amb The GIMP) i s'han afegit elements vectorials mitjançant xFig i Umlet e incorporats com PDF a la memòria. La memòria en format pdf s'ha generat amb l'eina pdflatex (inclosa amb teTeX). La bibliografia s'ha generat amb BibTeX. El procés de compilació s'ha automatitzat amb un *shell script* (i en fases inicials amb GNU Make).

El procés de executar l'algorisme sobre el joc de proves s'ha automatitzat amb *shell scripts* i s'han processat els resultats en taules LaTeX 2$_\varepsilon$ amb petits programes Java.

A.2 Desenvolupament de l'applet

L'*applet* s'ha desenvolupat principalment sobre Eclipse, tot i que inicialment fèiem servir joe/gVIM amb Make/Ant. S'ha fet servit un repositori de CVS pel control de versions, que posteriorment es va migrar a Subversion. El procés de compilació i empaquetat també està implementat amb Ant.

S'ha provat l'*applet* amb les màquines virtuals per Linux d'IBM i de Sun, i sota Windows amb la de Sun.

Els gràfics de l'*applet* s'han fet amb The Gimp.

B. Referència mode batch

Podem executar el programa en mode no interactiu per tal de facilitar l'extracció de resultats. Un exemple d'execució seria[1]:

```
java -jar sokoapplet.jar
batch=si
mapa=artificial-facil
solver=a*
situacio=simple
encallats=no
ordenacio=cost
heuristica=manhattansimple
```

El paràmetre `batch` especifica que volem execució no interactiva, i el paràmetre `solver` ens selecciona el mètode que farem servir per resoldre el mapa (`A*` és l'únic disponible). `mapa` escull el mapa d'entre els inclosos en l'aplicació[2].

El *solver* `A*`[3] necessita d'altres paràmetres addicionals:

`situacio simple` o `complexa`. La representació que farem servir (veure 3.1).

`encallats si` o `no`. Detectar blocs encallats o no (veure 3.4.1).

`ordenacio cost` cost (amplada prioritària), `dist` distància a l'objectiu (fa una cerca *best-first*), `costdist` cost més distància (A*). Veure 3.2.

[1] Hem introduït retorns de carro arbitraris. En la majoria de línies de comandaments, haurem d'introduir-ho tot en una sola línia

[2] El nom del mapa es correspon al títol del mapa sense espais

[3] De fet, aquest paràmetre engloba diferents cerques en arbre, no només un A*

`heuristica manhattansimple` o `distanciaempentes`. Veure 3.4.2.

També hem inclòs un petit *shell script* que invoca totes les combinacions possibles de configuració del programa per a un mapa. El podem invocar fent:

```
./test.sh nom-del-mapa
```

Resum

L'objectiu d'aquest projecte ha estat l'aprofundiment en un problema concret, el joc de Sokoban, que consisteix en un joc de planificació per a la consecució d'un objectiu conegut. Per dur a terme aquest estudi s'ha fet una implementació del joc amb una sèrie de mètodes configurables basades en algorismes de cerca com l'A* i amb l'ús de diferents heurístiques i altres tècniques específiques del joc. S'ha fet servir una metodologia evolutiva per a l'anàlisi i el disseny, i s'ha seleccionat el Java i Swing com a eines de codificació. S'ha dissenyat un conjunt de proves i s'han extret conclusions respecte de la complexitat d'aquest joc.

El objetivo de este proyecto ha sido la profundización en un problema concreto, el juego del Sokoban, que consiste en un juego de planificación para conseguir un objetivo conocido. Para llevar a cabo este estudio se ha hecho una implementación del juego con una serie de métodos configurables basados en algoritmos de búsqueda como el A* y el uso de diferentes heurísticas y otras técnicas específicas del juego. Se ha usado una metodología evolutiva para el análisis y el diseño, y se ha seleccionado Java y Swing como herramientas de codificación. Se ha diseñado un conjunto de pruebas y se han extraído conclusiones respecto a la complejidad de este juego.

The objective of this project has been an investigation of a specific problem, the Sokoban game, a planning game to achieve a known target. To carry out this study we have implemented a solver with a set of configurable resolution methods based on search algorithms like A* and using different heuristic functions and other domain-specific techniques. The development has followed an evolutive methodology for the analysis and design and using Java and Swing as the coding platform. A test set has been designed and we have drawn conclusions about the game's complexity.