SokoBOT: a Sokoban solver attempt

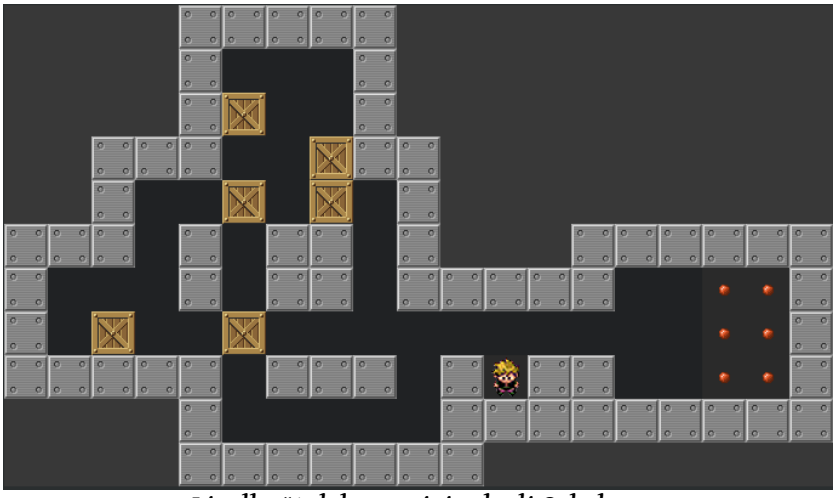
Progetto di Corso di Fondamenti di Intelligenza Artificiale

Indice

1 – Introduzione

Sokoban - traducibile approssimativamente con “magazziniere” - è un rompicapo creato e commercializzato in Giappone all’inizio degli anni ‘80. Il giocatore, che per semplicità - e affetto - chiamerò Sokoban da qui in avanti, si trova all’interno di un magazzino insieme ad un certo numero di casse che gli è consentito spingere, ma non tirare. Nel magazzino troverà degli specifici punti obiettivo presso cui dovrà spingere le casse, e il numero di tali obiettivi è pari a quello delle casse.

L’obiettivo del gioco, come facilmente intuibile, è spingere ognuna delle casse in un punto obiettivo, preferibilmente nel minor numero possibile di mosse. Un livello tipico del gioco è costruito come un labirinto in cui i muri sono disposti in modo da costringere Sokoban a valutare attentamente l’ordine e la direzione delle proprie “spinte” alle casse. Considerando che Sokoban non può attraversare né scalare i muri, in ogni momento può soltanto muoversi in una posizione vuota del livello oppure, se si trova in prossimità di una cassa, spingerla. Non gli è possibile spingere più casse “in fila indiana”: una spinta è possibile solo se, oltre la cassa che si vuole spingere, c’è uno spazio libero.



Livello #1 del set originale di Sokoban

Il set di regole del gioco è incredibilmente semplice. Probabilmente, se avessi potuto integrare un browser game in questo documento, non avrei nemmeno avuto bisogno di illustrarlo. Eppure, il problema computazionale ad esso sotteso è eccezionalmente complesso, e questo, unito alla facilità con cui si presta a fare da toy problem per modellare problemi reali di robotica e logistica, lo ha reso un grande classico nel campo della ricerca sull’IA. Dimostrabilmente, il problema di risolvere Sokoban appartiene alla classe di complessità NP-hard. Avrò modo di entrare nel dettaglio degli specifici elementi di difficoltà nella ricerca di soluzioni per un livello.

Questo lavoro di analisi del problema di ricerca sarà corredato da una suite di algoritmi risolutivi implementati in Java, integrati in un’applicazione grafica JavaFX che permetta di configurarli,

lanciarli e osservarne i risultati su un campione di livelli del gioco. L'obiettivo, nel complesso, è fornire una panoramica sulla modellazione del problema e sulla natura della sfida di creare un solver per Sokoban, rimarcando il modo in cui essa può essere ricondotta a tecniche domain-agnostic di ricerca su un grafo degli stati, ma anche illustrando alcune delle possibili ottimizzazioni domain-specific. Proverò inoltre a documentare in maniera frugale, evitando di perdere eccessivamente di generalità, il lavoro concreto di implementazione delle componenti del solver. Per ulteriori informazioni, si visiti il repo GitHub del progetto.

Concludo l'introduzione con un invito a temperare le proprie aspettative: risolvere Sokoban per un'IA, se non l'avessi già ricordato a sufficienza, non è una passeggiata al parco, e una esplorazione trasversale - e individuale - come quella che seguirà non è sufficiente per giungere a risultati solidi. Basti pensare che a Rolling Stones, il più celebre e performante Sokoban solver esistente, creato dall'University of Alberta, è stato necessario più di un decennio di sviluppo per riuscire a risolvere tutti i 90 puzzle del set standard di Sokoban. È interessante notare che un essere umano, data una quantità sufficiente di Red Bull e disdegno per la propria esistenza, potrebbe riuscirci in una giornata sola. Evidentemente l'IA non è ancora pronta a soppiantarci del tutto, almeno fin quando non imparerà a bere Red Bull e odiare se stessa.

Ad ogni modo, buona lettura e buona fortuna con la mia letale verbosità cagionata dalla noia del lockdown e dalle mie passate velleità letterarie. Rompendo momentaneamente la quarta parete e parlando direttamente ai docenti: ho incluso un indice proprio affinché vi sentiate liberi di cercare le parti che vi interessano maggiormente. Giuro che mi offendo solo un pochino.

2 – SokoBOT come agente intelligente

Questo capitolo ha lo scopo di formalizzare le caratteristiche di SokoBOT come agente intelligente, partendo dalla sua rappresentazione PEAS (Performance, Environment, Actuators, Sensors). In altre parole, sarà mostrato che un solver per Sokoban, pur operando nel contesto piuttosto astratto di un livello di gioco, può essere visto come un sistema che incamera una sequenza di **percezioni** su un **ambiente** e opera su di esso con una sequenza di **azioni** definite da una **funzione agente** in relazione a percezioni e azioni pregresse. L'operato dell'agente è guidato da una misura di **prestazione** che valuta il successo delle azioni compiute in base a criteri che saranno ampiamente discussi in seguito.

SokoBOT sarà un **agente basato su obiettivi** con lo scopo di trovare una sequenza di azioni di Sokoban che risolva un livello dato in input. Esso avrà contezza dello stato in cui si trovano se stesso e l'ambiente, nonché dell'effetto delle proprie azioni. Avrà memoria degli stati passati e compierà le proprie scelte, e dunque azioni, con l'unica intenzione di raggiungere uno stato riconosciuto come obiettivo. Le caratteristiche di uno stato obiettivo sono evidenti fin dal principio: SokoBOT tenderà verso uno stato che rappresenti una situazione di risoluzione del puzzle su cui sta operando. Questa tendenza a rappresentare la desiderabilità di uno stato in termini binari può, però, diventare più sfumata nel caso in cui il **programma agente** di SokoBOT basasse le sue azioni su criteri euristici che forniscano dei valori di utilità agli stati che ne rappresentino la desiderabilità, dove per stato *desiderabile* si intende *vicino alla soluzione*. In questo scenario, SokoBOT potrebbe essere interpretato come un ibrido tra un agente basato sugli obiettivi e un agente basato sull'utilità. Nonostante ciò, la natura stessa della "missione" di SokoBOT rende chiara la sua anima da *goal-based agent:* utilità degli stati o meno, esso verrà sempre incaricato di attivarsi, trovare una soluzione ad un puzzle e fermarsi.

2.1 – Ambiente e Sensori

L'ambiente in cui SokoBOT opera è il *maze* contenente il puzzle da risolvere o, più concretamente, una sua rappresentazione matriciale. Dato un qualsiasi livello di Sokoban, esso può essere visto come una griglia di caselle di gioco simile ad una scacchiera, rappresentato da una matrice avente alla posizione (i,j) un valore che identifica il contenuto della casella di gioco di riga i e colonna j . I valori possibili sono i seguenti:

- **E**: casella vuota, su cui è possibile spostare Sokoban o una cassa.
- **S**: casella occupata da Sokoban.
- **B**: casella occupata da una cassa.
- **W**: casella occupata da un muro; essa non è attraversabile e non cambierà mai valore.

Alcune delle caselle che nella rappresentazione iniziale del livello hanno valore E, S o B possiederanno un'etichetta aggiuntiva "**goal**", che permette di individuare le caselle obiettivo, ossia quelle su cui bisognerà spingere le casse. Possiamo immaginarla come un valore booleano, che viene registrato al momento della prima rappresentazione dell'ambiente di gioco e non è soggetto a cambiamenti nel tempo: le caselle obiettivo, difatti, possono cambiare contenuto ma non smetteranno mai di essere caselle obiettivo. Nell'implementazione concreta di SokoBOT, l'ambiente è codificato in un file .json contenente, come è possibile immaginare, una matrice di caratteri in cui, alla posizione (i,j), figura la lettera maiuscola corrispondente al contenuto iniziale della casella di gioco alla riga i e colonna j.

Codifica .json di un livello...

...e lo spazio di gioco corrispondente

La lettura dei file .json relativi ai livelli di gioco rappresenta, per SokoBOT, l'unico input richiesto per formare la conoscenza ambientale necessaria per risolvere il puzzle, e costituisce dunque l'unica sua istanza di "**sensore**". Parliamo ovviamente di un sensore software, che ha bisogno di entrare in attività una sola volta, al momento di percepire la configurazione matriciale iniziale del livello da risolvere. Da quel momento in avanti, l'agente non ha bisogno di ulteriori sequenze sensoriali. In ogni momento esso può inferire lo stato dell'ambiente sulla base delle azioni compiute su di esso a partire dalla prima percezione, operando in un ambiente **completamente osservabile** e **deterministico**. Questo schema percettivo *one-off* del programma agente di SokoBOT è reso possibile dal fatto che esso può **derivare** stati di gioco successivi al primo, completamente conscio sia delle caratteristiche del *maze,* sia delle conseguenze delle proprie azioni. In altre parole, le rappresentazioni degli stati successivi al primo possono essere sempre calcolate accuratamente, e non è utile introdurre nel modello dell'agente ulteriori percezioni. Del resto, la missione operativa del risolutore è meglio riassunta da:
"*Ecco un livello. Come si risolve?*"
piuttosto che da:
"Ecco un livello. Come si risolve? Ecco un livello leggermente diverso da quello di prima. Come si risolve? Ecco un altro livello ancora leggermente diverso. Come si risolve? ...".

Continuando con la classificazione dell'ambiente, i *maze* di Sokoban costituiscono un ambiente **semi-dinamico**: gli unici cambiamenti ad esso scaturiscono dall'intervento dell'agente risolutore, e a nessun poltergeist maligno è concesso di modificare la posizione degli elementi mentre SokoBOT è distratto, ma il passare del tempo incide sulla valutazione delle sue prestazioni. È un

ambiente **discreto** perché il numero di mosse ad ogni passo è limitato, e le possibili configurazioni delle caselle sono finite, pur tendendo spesso ad un numero altissimo.

Lo svolgimento innatamente singleplayer di Sokoban porta in genere a modellare un agente risolutore che operi in un ambiente **ad agente singolo**, e SokoBOT non fa eccezione. È opportuno ricordare, però, che in letteratura figurano vari tentativi, anche di discreto successo, di costruire risolutori modellati per agire su un ambiente multi-agente, interpretando le casse come agenti indipendenti l'uno dall'altro che "usano" Sokoban come strumento per raggiungere la loro meta, attirandolo a sé e dettandogli le spinte più convenienti.

2.2 – Attuatori e misure di Prestazione

Discorrere di *attuatori* nel contesto di un software di puzzle solving è certamente un formalismo. L'agente crea una rappresentazione degli stati dell'ambiente in forma di strutture dati ad hoc, e il kernel del sistema operativo che lo ospita gli concede – o meglio, per essere pedanti, concede alla JVM - il sommo potere di modificarle quandunque lo ritenga necessario per generare nuovi stati come conseguenza delle azioni scelte. L'influenza di SokoBOT sull'ambiente di gioco non deve fronteggiare particolari "attriti" e non deve mediare con altre componenti dell'ambiente. Deliberata una mossa di gioco, essa può essere semplicemente materializzata mandando in esecuzione i frammenti di codice che calcolano un nuovo stato e lo conservano in memoria.

Passiamo a qualcosa di decisamente più interessante. L'ovvia, minima e necessaria misura di prestazione di un agente risolutore di Sokoban è **risolvere Sokoban.** Scioccante, vero? Lo scopo dell'agente è produrre una sequenza di azioni che costituisca una soluzione per il puzzle in input, e se non riesce a fare ciò, nessun'altra metrica possibile ha importanza. Questo discorso va però commisurato alla difficoltà del problema che l'agente maneggia, ed essa dipende dalla taglia e dalla complessità del livello. Esistono strategie che assicurano la risoluzione di qualsiasi livello possibile, ma non necessariamente esse hanno costi sostenibili in termini di tempo di computazione e spazio di archiviazione.

Le altre misure di prestazione sono il **tempo di computazione**, la **memoria utilizzata** e il **numero di step** richiesti dalla soluzione generata per portare l'ambiente in uno stato vincente. Le tre misure citate vanno, idealmente, minimizzate, ma per la memoria utilizzata c'è da fare una precisazione. Lo scenario ideale non è strettamente usare *meno* memoria possibile, ma piuttosto di usarla *meglio* possibile. L'agente dovrebbe usare una quantità di memoria abbastanza bassa da non ritrovarsi privato delle risorse necessarie per restare in funzione, ma anche abbastanza alta da non dover scendere a compromessi computazionali non necessari che influenzino altre metriche di prestazione. In altre parole, vale il vecchio mantra: la memoria non usata è memoria sprecata.

Avere un tempo di computazione più basso possibile ha un'importanza piuttosto intuitiva, ma la definizione di *basso* può variare. Se intendiamo il risolutore come un'entità astratta che fornisce

una risposta a un enigma matematico, possiamo permetterci di aspettare diverse ore, perfino giorni o settimane, sebbene un tempo più breve possibile resti comunque decisamente preferibile. Se, invece, prendiamo sul serio il proposito di usare la ricerca su Sokoban come modello per problemi di robotica, dobbiamo considerare che non trarremmo particolare giovamento dal costruire un androide che, incaricato di spostare oggetti in determinati posti, si metta in moto una settimana dopo aver ricevuto il nostro ordine.

Infine, vale la pena di notare che esistono due modi diversi di intendere il *numero di step* richiesti da una soluzione: contare il numero totale di movimenti effettuati da Sokoban, oppure contare solo il numero di spinte alle casse. La questione è particolarmente rilevante perché adottare l'uno o l'altro come criterio principale porta la ricerca di una soluzione su strade sensibilmente diverse. Soluzioni che ottimizzano il numero di movimenti non faranno necessariamente lo stesso per il numero di spinte. Avrò modo di dettagliare abbondantemente questa distinzione più avanti.

3 – Sokoban come problema di ricerca

Come già accennato, il compito di SokoBOT non è triviale. Nella maggior parte dei casi, un livello non può essere risolto semplicemente considerando le casse una per una e spingendole verso una casella obiettivo. Anche perché, del resto, non sarebbe affatto divertente per un essere umano risolvere un livello del genere. È necessario solitamente *sbrogliare una matassa* di ostacoli e strettoie, prima di poter vedere chiaramente il percorso verso la vittoria. Per fare ciò, noi esseri umani possiamo impiegare una strategia d'insieme, valutare in maniera organica e flessibile la disposizione delle nostre risorse, e procedere immaginando il vantaggio o lo svantaggio delle varie strategie ideate dalle nostre sinapsi. L'obiettivo di questo capitolo è di introdurre gli ingredienti necessari per definire il problema di risolvere Sokoban con un agente risolutore che, essendo stato partorito da un wafer di silicone, non gode dei lussi cognitivi di cui sopra.

3.1 – Stati e transizioni

Per risolvere Sokoban computazionalmente, occorre vedere una partita come una serie di **transizioni** tra **stati**, determinate dal compimento delle azioni disponibili. L'insieme di tutti gli stati che è possibile raggiungere durante una partita è lo **spazio degli stati**, ed è il teatro dalle quinte immensamente larghe in cui un agente risolutore mette in scena la propria ricerca di una soluzione. Entriamo ora nello specifico dei singoli stati e delle transizioni tra essi. Possiamo dare due definizioni diverse di stato, che risulteranno familiari se si è prestata attenzione a quanto detto sui modi di *contare gli step* nel Par. 2.2.

Possiamo definire uno stato nello spazio di ricerca come una configurazione unica degli elementi semoventi dell'ambiente, ossia ognuna delle casse e Sokoban. Questa definizione implica che muovere Sokoban in qualsiasi modo, che sia spinta una cassa o meno, determina la genesi di un nuovo stato. Adottare questa definizione di stato porta a fissare il numero massimo di nuovi stati generabili in ogni momento a esattamente quattro, uno per ognuna delle quattro direzioni in cui Sokoban può muoversi.

Alternativamente, uno stato può essere definito come una configurazione unica delle casse. In questo caso, ignoriamo il libero girovagare di Sokoban e decidiamo che una configurazione sarà giudicata un nuovo stato soltanto quando una cassa sarà spinta e cambierà posizione. Due stati A e B possono essere considerati equivalenti se le casse sono nella stessa posizione, e se Sokoban può raggiungere, senza spingere casse, la posizione che occupa nello stato B partendo dalla posizione che occupa nello stato A e viceversa. Questa scelta aumenta il numero massimo di nuovi stati generabili a partire da un dato stato a quattro moltiplicato per il numero delle casse nel livello, poiché, al più, in un dato momento Sokoban può muovere ognuna delle casse in quattro direzioni diverse. Sebbene apparentemente si possa riscontrare una più rapida proliferazione degli stati, questa seconda definizione è la più utile, efficace e comune. Per darmi un tono, ho deciso di battezzare la prima definizione **move formulation** e la seconda **push formulation.**

3.2 – Formalizzazione del problema

Possiamo procedere a fornire, per ognuna delle formulazioni della definizione di stato, una descrizione formale completa del problema.

Move formulation

- **Stati:** l'insieme di tutte le possibili disposizioni discrete di Sokoban e delle casse all'interno del livello di gioco.
- **Stato iniziale:** il livello di gioco nel suo stato originale, ossia con casse e Sokoban disposti esattamente come prescritto dal file in input.
- **Azioni:** è possibile muovere Sokoban in ognuna delle quattro caselle adiacenti a quella in cui si trova nello stato corrente; notare che un'azione non ha nessun effetto se si sceglie di muovere Sokoban in una casella occupata da un muro o da una cassa che non può essere spostata. Formalmente, Azioni = {move_up, move_down, move_left, move_right}.
- **Modello di transizione:** una funzione che, dato uno stato di gioco *s* e una mossa di Sokoban *m*, associa alla coppia (s, m) un nuovo stato *s'* che rappresenti la configurazione del livello di gioco raggiunta applicando la mossa *m* allo stato *s*.
- **Test obiettivo:** si controlla se nello stato corrente tutte le casse sono su una casella obiettivo.
- **Costo di cammino:** il costo di una sequenza di azioni è pari alla cardinalità della sequenza stessa, ossia al numero di mosse compiute.

Push formulation

- **Stati:** l'insieme di tutte le possibili disposizioni discrete delle casse all'interno del livello di gioco.
- **Stato iniziale:** il livello di gioco nel suo stato originale, ossia con le casse disposte esattamente come prescritto dal file in input.
- **Azioni:** è possibile spingere le casse verso ognuna delle quattro caselle adiacenti a quella occupata da tale cassa nello stato corrente; notare che un'azione non ha nessun effetto se si sceglie di spostare la cassa in una casella occupata da un muro o da un'altra cassa. Formalmente: foreach cassa c -> Azioni_c = {c.push_up, c.push_down, c.push_left, c.push_right}
- **Modello di transizione:** una funzione che, dato uno stato di gioco *s,* una cassa c e una mossa di spinta *m*, associa alla tripla (s, c, m) un nuovo stato *s'* che rappresenti la configurazione del livello di gioco raggiunta a partire dallo stato *s* applicando la mossa *m* alla cassa *c*.
- **Test obiettivo:** si controlla se nello stato corrente tutte le casse sono su una casella obiettivo.
- **Costo di cammino:** il costo di una sequenza di azioni è pari alla cardinalità della sequenza stessa, ossia al numero di spinte compiute.

Si noti che un agente risolutore che adotti la push formulation produrrà, almeno in linea teorica, una sequenza di spinte alle casse e non una sequenza di mosse di Sokoban. L'implementazione della push formulation prodotta per il programma agente di SokoBOT, però, si curerà di *unire i puntini* tra una spinta e l'altra, mantenendo così la possibilità di avere la soluzione nella forma più pratica e intuitiva possibile, cioè come sequenza di azioni di Sokoban.

3.3 – Schemi di espansione dei nodi nell'albero di ricerca

Il compito di cercare uno stato desiderabile in un sistema di stati e transizioni si configura convenzionalmente come una ricerca su un albero implicito, di cui possediamo soltanto il nodo radice che, nel caso di Sokoban, rappresenta lo stato di partenza del puzzle. Durante l'esecuzione di un algoritmo di ricerca, i nodi visitati, a partire dalla radice, sono **espansi**: vengono "scoperti" e aggiunti all'albero tutti i nodi adiacenti a un nodo appena visitato. In termini di ricerca tra stati, questo ci permetterà di esplorare tutti gli stati a una sola transizione di distanza dallo stato in cui ci troviamo. Logica e ordine di tale esplorazione dipendono dall'algoritmo scelto, ma l'idea è sempre quella di procedere, espansione dopo espansione, ad esplorare nodi generati fin quando non si giunga ad esplorare un nodo che rappresenti uno stato obiettivo. Uno stesso stato nello spazio può essere raggiunto con sequenze di azioni diverse tra di loro, e il risultato è che durante la ricerca costruiremo un **albero con cammini ciclici**, che presenta cioè cammini con più copie dello stesso nodo. Come è facile immaginare, questa caratteristica è un elemento problematico, che può far esplodere la complessità di tempo e di spazio della ricerca, se non affrontato. Breve nota: a seguire, conterò sul fatto che il lettore abbia conoscenze pregresse sulla teoria dei grafi e sugli algoritmi di ricerca su grafi.

Torniamo sulle definizioni di stato, stavolta consci del fatto che rappresenteremo gli stati come nodi di un albero di ricerca. Definire gli stati, e dunque i nodi di ricerca generati, in base alle spinte alle casse porta il **branching factor** dell'albero a $4 \times b$, dove b è il numero delle casse presenti nel livello. Ma definire gli stati in base alle mosse totali di Sokoban, di contro, aumenta la **profondità massima** della ricerca poiché, generalmente, le soluzioni prevedono un numero ottimale di mosse ben superiore al numero ottimale di spinte. La questione è rilevante perché la profondità del percorso di ricerca tende, negli algoritmi di ricerca su grafi, ad avere un peso computazionale maggiore rispetto al branching factor. Ecco spiegato il motivo per cui è generalmente preferibile adottare la push formulation rispetto alla move formulation. Espandere i nodi in base alle mosse porta a generare un numero molto più alto di stati, la maggior parte dei quali non sono affatto utili nella ricerca della soluzione: valutare anche la possibilità che Sokoban se ne vada a zonzo per il livello senza curarsi delle casse che è pagato per spingere può essere un'idea gradevole per i procrastinatori, ma di certo non avanza la quest dell'agente risolutore.

Nell'implementazione concreta di SokoBOT sarà presente la possibilità di utilizzare entrambi gli schemi di espansione dei nodi per osservarne l'efficacia. Al tempo della stesura di questo paragrafo, sono perlopiù entrambi pronti e testabili ed è già evidente che con l'espansione in base

alle mosse si incorre in grandi difficoltà già in puzzle di complessità moderata, con l'esplosione incontrollabile e indesiderabile del numero degli stati, specialmente in livelli "aperti", con tanto spazio a disposizione per i peregrinaggi infruttuosi di Sokoban.

Spezzando una lancia in favore della move formulation, c'è da dire che effettuare la ricerca della soluzione espandendo i nodi in base alle possibili spinte delle casse non può garantire l'ottimalità della soluzione in termini di mosse totali, ma solo in termini di spinte totali. Peraltro, implementare uno schema di espansione dei nodi che assecondi la move formulation risulta molto più semplice e naturale: se ci troviamo in uno stato, per espanderlo e valutare la sua progenie non dobbiamo fare altro che eseguire tutti e quattro i movimenti disponibili e immagazzinare gli stati così prodotti in nuovi nodi. Terremo conto, ovviamente, del fatto che potrebbero esserci ostacoli intorno a Sokoban, e dunque alcune delle mosse potrebbero non essere effettuabili.

Passando alla push formulation, la situazione è più complessa sia concettualmente che computazionalmente. Per generare gli stati adiacenti a quello corrente, dovremo provare tutte le spinte a una cassa possibili. Ciò significa, innanzitutto, individuare le caselle che contengono una cassa, poi procedere, una cassa alla volta, nel seguente modo:

1. Si individuano le caselle libere adiacenti alla cassa. Una casella è selezionata solo se dal lato opposto ad essa, oltre la cassa, c'è una casella libera che possa ospitare la cassa dopo la spinta. Le caselle che soddisfano tale condizione saranno chiamate **caselle di spinta**. Si noti che ogni cassa potrebbe essere disponibile per più spinte diverse, fino ad un massimo di quattro.
2. Si verifica che esista un percorso nel maze tra Sokoban e le caselle di spinta individuate nel passo precedente. Questo percorso dovrà essere praticabile senza la necessità di spostare altre casse. Nel caso non fosse possibile per Sokoban raggiungere una delle caselle di spinta, una particolare spinta non sarà effettuabile, e si procederà a valutare la prossima casella.
3. Se invece quel percorso esiste, lo si determina passo per passo, curandosi che sia il più breve percorso possibile tra Sokoban e la casella da raggiungere.
4. Infine, si muove Sokoban lungo il percorso individuato e si esegue la spinta, generando un nuovo stato secondo la push formulation.

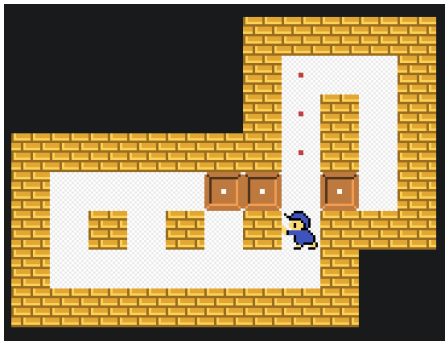
I passi 2 e 3, pur concettualmente distinti, sono stati implementati in congiunzione in SokoBOT, immaginando le caselle di gioco come un – ulteriore – grafo in cui caselle vuote adiacenti siano unite da archi e lanciando su di esso una **ricerca BFS**. La ricerca si arresta con un successo quando la casella di spinta che cercavamo viene trovata: la natura della ricerca in ampiezza ci garantisce che siamo arrivati al nostro obiettivo con il numero minimo di passi. Ovviamente, se arrivassimo ad aver esplorato tutte le caselle vuote senza trovare l'obiettivo, concluderemmo che la casella cercata non era raggiungibile.

Per informazioni più specifiche sull'implementazione degli schemi di espansione in SokoBOT, fare riferimento al codice nel repo GitHub. Ho provato a commentarlo copiosamente, proprio per poter evitare di scendere in dettagli implementativi in questa sede.

3.4 – Elementi di difficoltà nella ricerca

Come già accennato in varie occasioni, lo spazio degli stati su cui operare la ricerca ha una taglia importante. Assumendo di espandere i nodi durante la ricerca per spinte e non per mosse, sia il **fattore di branching** che la **profondità della soluzione** all'interno del grafo di ricerca tendono ad assumere valori molto alti. Nei 90 puzzle del set standard di Sokoban il fattore di branching più alto incontrato ammonta a 136, con una media pari a 12, mentre la profondità della soluzione varia tra 97 e 674. Inoltre, l'interezza dello spazio di ricerca conta, in media, 10^{18} stati distinti, rendendo chiara l'esigenza di imboccare quanto prima una direzione che indirizzi la ricerca su percorsi che abbiano probabilità migliori di terminare con la soluzione. Una ricerca completa in questo spazio degli stati, visitando cioè tutti i nodi per ciascun livello di profondità fino al raggiungimento della soluzione, può risultare esosa in termini di tempo e/o di spazio.

Oltre all'intrattabilità meramente numerica, la natura stessa del gioco introduce un ulteriore ostacolo: la presenza di **deadlock**, configurazioni di stallo in cui risulta impossibile, continuando la partita, arrivare a vincerla. Il dramma però non consiste nella loro esistenza, quanto nella difficoltà di rilevarli e interrompere il branch corrente della ricerca in caso se ne incontri uno. A volte, è difficile perfino per un essere umano accorgersi di aver disposto le casse in una maniera che annulla irreversibilmente le chance di vittoria.



Esempio di una situazione di deadlock

Aver creato una configurazione di deadlock non significa aver creato una condizione in cui non sia più possibile muovere Sokoban o le casse. Senza l'ausilio di meccanismi ad hoc di *deadlock detection,* non si ha nessuna possibilità di accorgersi del vicolo cieco e tagliare il ramo corrispondente della ricerca. Al contrario, si continuerà a valutare movimenti e spinte che non possono più portare alla soluzione. Esistono, per onor del vero, casi in cui, almeno se si adotta la push formulation, avviene automaticamente una sorta di deadlock detection. Nel caso in cui spingessimo le casse in degli angoli del livello, esse non potranno più essere spostate, e dunque quel percorso di ricerca terminerà. Ovviamente si tratta di un *fringe case*: i deadlock più comuni durante una partita sono molto più sottili e difficili da rilevare.

4 – Risolvere Sokoban

Introdotto il problema di ricerca in generale, passiamo a stabilire come un Sokoban solver può procedere per risolverlo, e come SokoBOT, nella fattispecie, procede per risolverlo. La trattazione dell'argomento sarà incentrata sui metodi di risoluzione che sono – o saranno – implementati concretamente in SokoBOT, ma non mancheranno riferimenti ad altre strategie e ottimizzazioni possibili e documentate. Si noti che molte specifiche riguardanti concetti generali di ricerca informata e non informata saranno date per scontate. Non è particolarmente d'interesse, in questo contesto, fornire una panoramica dettagliata su strategie trasversali di graph search, che saranno toccate *en passant*.

4.1 – Rappresentazione e trasposizione degli stati

Durante la ricerca, SokoBOT usa oggetti *Node* dedicati alla rappresentazione di singoli stati nello spazio di ricerca. Per maggiore praticità nel processing dei dati essi comprendono, oltre alla matrice che tiene traccia del contenuto di ognuna delle caselle di gioco – incapsulata in un oggetto *GameBoard* -, una pletora di informazioni sia sullo stato rappresentato che sul nodo nell'albero di ricerca, ad esempio un riferimento diretto alle caselle che contengono le casse, oppure il livello di profondità nell'albero in cui il nodo è stato trovato. La classe *Node* non è un POJO inerte, ma implementa anche la logica delle operazioni sui nodi che gli algoritmi di risoluzione dovranno effettuare, tra cui l'espansione di un nodo.

In maniera del tutto trasversale alla strategia di ricerca adottata, è necessario effettuare una prima, essenziale forma di pruning dello spazio degli stati, ossia impedire la nascita di **cammini ciclici** nell'albero di ricerca. In altre parole, è necessario evitare che l'espansione di un nodo porti a generare nodi che rappresentino stati già esaminati, portando lo spazio di ricerca a crescere in maniera incontrollabilmente superiore rispetto all'effettivo spazio degli stati. Senza questa basilare accortezza, anche risolvere livelli estremamente piccoli diventa un piccolo miracolo computazionale. Fortunatamente, è relativamente facile ottenere il risultato sperato implementando una struttura dati che memorizzi delle **trasposizioni** degli stati toccati, ossia delle rappresentazioni sintetiche e – se tutto va bene – univoche di tali stati. Ogni volta che un nuovo nodo viene generato durante un'espansione, si controlla se la sua trasposizione è già presente: in quel caso, possiamo potare un ramo; nel caso in cui non fosse presente, invece, possiamo procedere esplorando anche in quella direzione, certi del fatto che stiamo rappresentando uno stato inedito. Ovviamente, ad ogni nuova "accettazione" di un nodo espanso, dobbiamo inserirne la trasposizione nella struttura.

Le trasposizioni in SokoBOT sono implementate trasformando la board di gioco in un array di byte e producendone il valore hash MD5, memorizzato in una variabile long e inserito nella struttura delle trasposizioni. L'unica componente ad essere coinvolta nell'hashing è la matrice di celle che rappresentano uno stato della board di gioco, mentre i campi "di servizio" sono ignorati per evitare comportamenti indesiderati e sprechi di spazio. Nodi che possiedono board di gioco in configurazione identica avranno una trasposizione identica: basterà dunque consultare la lista di tali trasposizioni per verificare se abbiamo già incontrato un stato. L'algoritmo di hashing MD5, pur non essendo crittograficamente sicuro da decenni, ha una resistenza alle collisioni che lo rende più che sufficiente per evitare che due stati diversi siano trasposti con il medesimo valore hash.

Si noti che, nel caso si stiano espandendo i nodi in funzione delle spinte alle casse, non sarà sufficiente trasporre la board di gioco nell'esatto stato in cui la si trova. Agendo in questo modo, infatti, trasporremmo gli stati tenendo conto della posizione di Sokoban. Due trasposizioni, anche presentando le casse nella stessa posizione, sarebbero comunque ritenute uniche nel caso Sokoban figurasse in posizioni diverse, determinando l'esplorazione di stati che, per la push formulation, non sarebbero *nuovi* stati. Per rimediare, possiamo omettere Sokoban dalla rappresentazione prima di effettuare l'hashing. Dobbiamo però accertarci che, nel confrontare le trasposizioni di due stati A e B con la stessa configurazione delle casse, questi siano ritenuti identici solo se esiste un percorso tra la posizione di Sokoban in A e la posizione di Sokoban in B e viceversa. Il modo più semplice per assicurare tale condizione è codificare le trasposizioni in modo da prendere in considerazione la board priva di Sokoban, con l'aggiunta della lista ordinata delle caselle raggiungibili da Sokoban in tale configurazione. Trasposizioni congruenti in tale codifica assicurano di rappresentare solo stati congruenti secondo la push formulation.

4.2 – Algoritmi di ricerca non informata

Risolvere Sokoban lanciando una ricerca cieca senza sfruttare conoscenza aggiuntiva per orientare la navigazione nell'albero di ricerca potrebbe sembrare futile. In effetti, gli effort di maggiore successo sono storicamente basati su algoritmi di ricerca informata. Gli algoritmi di ricerca non informata hanno però il merito intrinseco di non aver bisogno di una stima euristica per produrre risultati. Sebbene appaia sciocco non ottimizzare la ricerca con criteri euristici laddove fossero disponibili, occorre ricordare che le valutazioni euristiche non piovono sempre dal cielo. La loro elaborazione potrebbe essere computazionalmente intensa, e potrebbe richiedere l'uso di strutture dati aggiuntive che peggiorano l'occupazione di memoria. Ad esempio, proprio la stima euristica "corretta" della bontà di uno stato in Sokoban, che descriverò in seguito, non è delle più semplici da calcolare, e domina il tempo d'esecuzione degli algoritmi di ricerca informata. La ricerca non informata aumenta la fetta disponibile del budget computazionale tagliando i costi di calcolo al momento di espandere i nodi.

SokoBOT può avvalersi di una implementazione semplicissima e assolutamente convenzionale di una **ricerca in ampiezza**, con le consuete caratteristiche: ottimalità nel caso in cui il costo di

cammino cresca insieme alla profondità dei nodi, completezza, e un investimento potenzialmente inaccettabile di memoria. Abbinato a una struttura di trasposizioni efficace, esso è perfettamente sufficiente per trovare la soluzione ottima a puzzle di taglia moderata, e mi è stato dunque utilissimo per testare l'impalcatura software del solver prima di iniziare a perfezionare le meccaniche di risoluzione.

È interessante notare che, se si fosse implementata una ricerca a costo uniforme, la natura dell'espansione di nodi nello spazio di ricerca di Sokoban l'avrebbe fatta collassare in una ricerca in ampiezza. Nel nostro problema di ricerca, ogni volta che si espande un nodo la sua prole rappresenta tutti gli stati raggiungibili con una singola mossa o spinta. L'ovvia conseguenza è che tutti i nodi ad un certo livello di profondità avranno lo stesso costo di cammino. Una ricerca a costo uniforme non farebbe altro che esplorare tutti i livelli di profondità uno dopo l'altro, visto che non troverà mai un nodo in un livello che abbia costo inferiore ai propri compagni di livello, e allo stesso modo non troverà mai, per qualsiasi $k>0$, un nodo al livello $n+k$ che abbia costo minore o uguale rispetto a qualche nodo al livello n. In tal modo, non "sfalserà" mai il processo di ricerca e procederà esattamente come se da grande avesse sempre sognato di fare la BFS.

4.3 – Euristiche di stima

Un modo piuttosto comune di migliorare le prestazioni di un agente risolutore impegnato in imprese complesse è *informare* la ricerca individuando un criterio euristico per stimare la bontà di uno stato. Esso sarà distillato matematicamente in una **funzione euristica** che ci fornisca, dato un qualsiasi stato, una stima più o meno accurata della sua distanza dalla soluzione.

Consideriamo di trovarci all'interno di uno dei maze di Sokoban, e di valutare i passi compiuti in termini di spinte alle casse (push formulation). Quanto siamo distanti dalla soluzione? La risposta sarà, in qualche modo, una funzione della distanza delle casse dagli obiettivi o, per essere più precisi, della **Manhattan distance** delle casse dagli obiettivi, ossia la distanza tra due punti in un piano in cui, esattamente come negli isolati newyorkesi, ci si può muovere solo lungo una griglia. Ora, dobbiamo formalizzare una funzione euristica **ammissibile** – che fornisca, cioè, un lower bound della soluzione - usando le distanze tra le casse e gli obiettivi.

Possiamo vedere casse e obiettivi come le due partizioni di un grafo bipartito, e creare un **matching** che unisca ogni cassa all'obiettivo ad essa più vicino, dopodiché sommare le Manhattan distance calcolate tra i membri delle coppie così formate. La funzione appena descritta è sicuramente un'euristica ammissibile, ma è abbastanza facile notare che può sottostimare brutalmente la distanza dalla soluzione: è un lower bound molto largo. Se due o più casse si trovano in prossimità di uno stesso obiettivo, entrambe saranno "matchate" con esso, ma solo una cassa può occupare ciascun obiettivo, e quindi il reale costo della soluzione sarà probabilmente molto più alto. Avremmo una sottostima quasi imbarazzante nel caso estremo in cui tante casse fossero vicine ad un solo obiettivo e tutti gli altri obiettivi si trovassero molto distanti.

È chiaro che il matching tra le due partizioni debba essere perfetto: a ogni cassa deve essere associato un obiettivo diverso. La soluzione matematicamente approvata al nostro dilemma è quindi calcolare il **matching perfetto di valore minimo**, ossia quello che minimizza la somma delle Manhattan distance delle coppie scelte, in modo da garantire che tale matching perfetto sia anche dimostrabilmente un lower bound della soluzione. Tale calcolo è la specialità dell'**algoritmo ungherese**, anche noto come algoritmo di Kuhn-Munkres. Si tratta di un algoritmo di ottimizzazione matematica che opera su sistemi di vincoli esprimibili in forma matriciale, ma può essere espresso in termini di assegnazioni su grafi bipartiti. Il tempo d'esecuzione è polinomiale, ma non particolarmente veloce – $O(n^4)$ -, determinando, sulle migliaia di esecuzioni richieste per valutare gli stati toccati da un algoritmo di ricerca informata, un sensibile overhead. Sarà possibile testare tale algoritmo di stima in SokoBOT, così come l'euristica naïf di assegnazione minima greedy descritta in precedenza.

4.4 – Algoritmi di ricerca informata

Stabilita un'euristica di valutazione – o un array di euristiche da sfruttare in una qualche combinazione – segue solitamente l'implementazione di un algoritmo che possa trarne vantaggio. La scelta è ricaduta su **A***. Esso, seppure stravolto da decine di ottimizzazioni domain-specific, è alla base del già citato *Rolling Stones* della University of Alberta, ed è generalmente considerato una scommessa sicura. SokoBOT implementa una ricerca A* canonica: ottimale e completa data un'euristica ammissibile come quelle descritte.

Partendo da A*, è bastato modificare la formula di assegnazione delle etichette da $g(n)+h(n)$ al solo $h(n)$ per ottenere un algoritmo greedy di **ricerca Best First.** Escludendo il costo di cammino al momento di inserire un nodo nella coda a priorità che memorizza la frontiera, si sta abiurando qualsiasi garanzia matematica di completezza e ottimalità**.** In compenso, però, questo tipo di ricerca greedy è in grado di trovare, per molti livelli, una soluzione subottimale in tempo particolarmente ridotto, tendendo in maniera "famelica" verso stati vicini alla soluzione senza curarsi di quanto sia lungo il cammino per giungervi.

Come *tie breaker* per definire l'ordine delle visite tra nodi con la stessa stima euristica, è stato usato un semplice schema di **move ordering** definito *per inerzia*. Esso favorisce i nodi che contengono una spinta ad una cassa che era stata toccata anche dal nodo genitore. Una soluzione ad un puzzle, infatti, comprende molto spesso lunghe serie di spinte alla stessa cassa. Ordinare le visite in questo modo può dunque migliorare le prestazioni dell'algoritmo per alcuni livelli.

Infine, si è valutata anche una versione memory-bounded di A*, ma la sua utilità si è rivelata piuttosto ridotta. Il bound di memoria entra in gioco solo in livelli fuori dall'immediata portata di questo solver, che trovano nel tempo di esecuzione un limite molto più evidente dell'occupazione di memoria. Un solver implementato con maggiori accortezze low-level, magari parallelizzando le operazioni dove possibile, avrebbe potuto forse trarne giovamento, specialmente se testato su hardware di caratura maggiore.

4.5 – Algoritmi ad approfondimento iterativo

Tra gli algoritmi implementati si troveranno delle versioni ad **approfondimento iterativo** di una **DFS** e di **A***, ma i loro risultati non sono soddisfacenti. Al di là della penalità in prestazioni intrinseca - e comunque irrilevante a livello asintotico - dell'iterative deepening, la natura stessa del nostro specifico problema di ricerca porta in seno alcune difficoltà per IDDFS e IDA*.

Se usati per risolvere un livello di Sokoban, la DFS e gli algoritmi *DFS-like* come IDA* non possono garantire l'ottimalità della soluzione (e nemmeno una accettabile sub-ottimalità) se non rilassando il vincolo di non esplorazione dei nodi già trasposti. Il motivo è piuttosto semplice. Sia *s* uno stato che compare nelle fasi iniziali della soluzione ottima, e sia *N* l'insieme di tutti i nodi nello spazio di ricerca che rappresentano *s*. Immaginiamo che la ricerca in profondità, nel valutare uno dei primi rami, si imbatta in un nodo appartenente ad *N* a un livello molto basso, dopo aver fatto molta strada. Quando si farà backtracking e si giungerà sul ramo che, avendo un nodo in *N* tra i primi nodi visitati, ci condurrebbe alla soluzione ottima, tale nodo non sarà espanso perché rappresenta uno stato già trasposto. Non ci sono garanzie che una soluzione della stessa qualità possa essere trovata, ma possiamo comunque essere sicuri che, partendo dal punto in cui ci siamo imbattuti per la prima volta in un nodo che rappresenta *s,* una soluzione sarà comunque scovata. In buona sostanza, **manteniamo la completezza ma perdiamo l'ottimalità**, ricalcando le caratteristiche di una DFS *vanilla* piuttosto che quelle di una DFS ad approfondimento iterativo.

Potremmo permettere all'algoritmo di "ricordare" soltanto le trasposizioni dei nodi presenti in memoria, ossia tutti i nodi sul percorso tra la radice e il nodo da valutare. Nel momento in cui si fa backtracking - ossia all'uscita dalla chiamata ricorsiva – la trasposizione dei nodi valutati sarà cancellata. Il problema di questo approccio è che, naturalmente, non potremmo più fare pruning in presenza di nodi che rappresentano stati già incontrati in percorsi diversi da quello corrente, e ciò renderebbe gli algoritmi con approfondimento iterativo lentissimi e praticamente inutili in questo contesto.

Esiste una possibile mitigazione sicuramente più efficace, che è quella implementata in SokoBOT. È possibile modificare la struttura delle trasposizioni in modo da registrare, oltre alla trasposizione stessa, a che profondità nell'albero di ricerca ci si trovava quando si è trasposto uno stato. Preso in esame un nodo che rappresenti uno stato già trasposto, esso sarà espanso solo se l'ultima trasposizione di quello stato era stata effettuata ad una profondità maggiore di quella corrente. Questa ottimizzazione risulta efficace nel conservare l'ottimalità degli algoritmi in questione, ma è facile notare che le prestazioni sono comunque severamente limitate dalla alta probabilità di valutare più volte uno stesso stato a profondità diverse dell'albero di ricerca durante una ricerca in profondità. Almeno nella loro forma canonica, tali algoritmi non sono tra i più adatti al nostro scopo, ma sarà comunque possibile testarli nel programma agente.

4.6 – Deadlock detection

L'individuazione delle situazioni di deadlock è la montagna più alta da scalare per un solver di Sokoban, umano o artificiale che sia. I potenziali **pattern** di deadlock sono in numero elevatissimo, e stabilire dinamicamente durante la ricerca se una certa configurazione costituisca un deadlock o meno è un problema tanto difficile quanto risolvere il livello. È per questo che cercare un algoritmo che semplicemente scandagli la griglia di gioco e scovi deadlock è una strada troppo impervia, su cui nessuno osa metter piede.

4.6.1 – Pattern database

La scelta più comune è creare un **database** di configurazioni di stallo da approntare prima dell'inizio della ricerca e consultare staticamente. Dopo ogni spinta, si isola dal livello una serie di sottomatrici orientate lungo la direzione della spinta e le si confronta con i pattern di deadlock già noti presenti nel database. Naturalmente, si procede a scartare il nodo corrente se vi si identifica un deadlock. Anche questa, però, è una gran bella impresa. Precomputare griglie $n \times m$ di deadlock specifici per un livello richiede moltissima potenza di elaborazione. Precomputare griglie di deadlock generici ne richiede ancora di più. Perdipiù, per arrivare a livelli di pruning apprezzabili su qualsiasi livello, c'è bisogno di tenere in memoria milioni di configurazioni di deadlock, il che è fuori dalla portata di workstation casalinghe.

Rolling Stones fa affidamento proprio su un grosso database di deadlock precomputati, ma riprodurre una strategia del genere sarebbe decisamente troppo dispendioso e ben fuori dallo scope di questo progetto. Per onor del vero, ho implementato una piccola *lookup table* che contiene tutti i deadlock pattern inscrivibili in una sottomatrice 2×2 e alcuni dei deadlock pattern inscrivibili in una sottomatrice 3×3. L'effetto dei pruning determinati da questo minuscolo database di pattern è ridotto ma misurabile. In ogni caso, la mia attenzione si è rivolta principalmente verso alcuni meccanismi "concettuali" di deadlock detection, che descriverò a seguire.

4.6.2 – Posizioni morte

Una posizione di gioco p è *morta* quando non è possibile spingere una cassa che si trovi in p in nessuna delle caselle obiettivo, indipendentemente dallo stato delle restanti casse. Spostare una cassa in una posizione morta è condizione sufficiente per avere un deadlock. Per testare se una cassa si trova in una posizione morta basta eliminare tutte le altre casse dal livello e controllare se è possibile spingere l'unica cassa rimanente in una qualsiasi delle caselle obiettivo. Si tratta di un'operazione piuttosto semplice e computazionalmente indolore, ma estremamente utile. Nonostante i deadlock determinati da posizioni morte siano una percentuale microscopica di

quelli possibili, essi sono tra i più frequenti. Questa accortezza piuttosto basilare, dunque, può determinare un grosso guadagno in prestazioni. Oltretutto, poichè le posizioni morte non variano durante la partita, è possibile calcolarle una volta sola, prima di iniziare la ricerca della soluzione.

Sotto il cofano, il deadlock detector di SokoBOT crea una versione priva di casse del livello di gioco. Dopodiché considera una per una le caselle vuote del livello: per ognuna di esse, vi posiziona una cassa e avvia una rapida **Best First Search** dotata di una struttura di trasposizioni dedicata. Si considera come stato di arrivo un qualsiasi stato in cui tale cassa fosse posizionata su una casella obiettivo. L'euristica che guiderà la ricerca sarà semplicemente la Manhattan distance dall'unica cassa presente all'obiettivo più vicino ad essa. Nel caso in cui non fosse trovata una sequenza di spinte che termina con la cassa in un obiettivo, la casella da cui eravamo partiti sarà dichiarata una posizione morta attraverso un campo booleano. Durante la ricerca della soluzione, basterà un banale controllo di quel campo ogni volta che una cassa verrà spostata per individuare qualsiasi deadlock generato da posizioni morte.

4.6.3 – Casse congelate

Un criterio leggermente più complesso per individuare parte dei possibili deadlock è cercare, dopo ogni spinta, casse che siano *congelate,* ossia impossibili da muovere. Si parla, in quel caso, di **freeze deadlock.** Una cassa è congelata quando è bloccata sia sull'asse orizzontale che sull'asse verticale, ossia se, per entrambi gli assi, sussiste almeno una delle seguenti condizioni:

- La cassa è posizionata in prossimità di un muro.
- La cassa è posizionata tra due posizioni morte.
- La cassa è posizionata in prossimità di una cassa congelata.

Poniamo di voler certificare il "congelamento" di una cassa *c.* Mentre le prime due condizioni sono facilmente verificabili con un check sulle caselle adiacenti a quella occupata da *c*, per verificare la terza occorre richiamare ricorsivamente la procedura di controllo su casse eventualmente posizionate in prossimità di *c*.

Ovviamente, ogni cassa su cui viene propagato il controllo può a sua volta dover *passare la palla* a un'altra cassa, al punto che, se *c* si trova in un gruppetto di casse addossate, esse saranno tutte controllate. Dopo i primi due check, è opportuno trasformare momentaneamente la cassa in un muro, per impedire che lo stack di chiamate ricorsive a casse attigue si propaghi indefinitamente: in questo modo, tutte le casse interessate da un possibile freeze deadlock vengono controllate esattamente una volta. Per brevità, ometterò tale ottimizzazione dallo pseudocodice esemplificativo a seguire.

```
Function: boolean <- checkFrozen (box)
    variable: boolean isFrozen

    //controlliamo le prime due condizioni orizzontalmente
    if isWall (right (box)) OR isWall (left (box))  → isFrozen = true
    else if isDeadPosition (right (box)) AND isDeadPosition (left (box))  → isFrozen = true
    else → isFrozen = false
    //se non abbiamo individuato freeze orizzontali con le prime due condizioni
    //procediamo a controllare la terza avviando la catena di chiamate ricorsive
    if isFrozen = false
        if checkFrozen (right (box)) OR checkFrozen (left (box)) → isFrozen = true
    //se isFrozen è ancora falsa significa che non abbiamo individuato casse confinanti
    //bloccate, e se tutte e tre le condizioni sono false, l'asse orizzontale è libero e dunque box non è congelata
    if isFrozen = false → return false
    //se arriviamo qui, l'asse orizzontale è bloccato: procediamo con gli stessi passi
    //descritti prima anche per l'asse verticale per stabilire definitivamente se la cassa è bloccata
    if isWall (up (box)) OR isWall (down (box))  → isFrozen = true
    else if isDeadPosition (up (box)) AND isDeadPosition (down (box))  → isFrozen = true
    else → isFrozen = false

    if isFrozen = false
        if checkFrozen (down (box)) OR checkFrozen (up (box)) → isFrozen = true

    return isFrozen
```

4.7 – Idee inesplorate

Presento brevemente alcune delle tecniche di risoluzione citate in letteratura ma non prese in considerazione per un'integrazione immediata in SokoBOT, per la loro impraticità di implementazione o per l'utilità estremamente specifica.

Una delle idee più strampalate e contemporaneamente più interessanti riguardo al Sokoban solving è la possibilità di risolvere i livelli **al contrario**. Si parte con le casse già posizionate sugli obiettivi e si tenta di portarle nelle posizioni prescritte dallo stato iniziale del livello. Naturalmente, in questo caso Sokoban non dovrà spingere le casse, ma **tirarle**, in modo che, ribaltando la soluzione generata con questo processo, si ottenga una soluzione legale al puzzle di partenza, espressa in termini di spinte.

Considerare questa possibilità apre la possibilità di applicare algoritmi di ricerca **bidirezionale**, con il potenziale guadagno che ne consegue in termini di tempo di esecuzione. Ciò nondimeno, nessuno ci vieta di implementare algoritmi puramente **inversi**, che procedano in maniera classica,

ma a senso invertito. Il vantaggio di questo approccio riguarda la natura dei *pull deadlock* se confrontati ai *push deadlock*. Risolvere un puzzle tirando le casse implica che, laddove si riscontri un deadlock, ci accorgeremo che in molti casi Sokoban si è "murato" in un angolo tirando a sé le casse. Si tratta di deadlock che non pesano sull'albero di ricerca, perché lasciano ben poche mosse a disposizione di Sokoban.

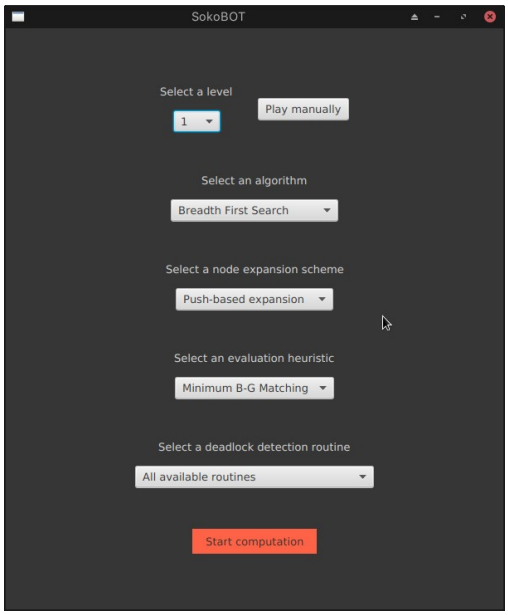
Alcuni solver fanno uso di tecniche di ottimizzazione avanzata dette **macro**. Una macro individua configurazioni in cui una certa sequenza *scriptata* e predeterminata di mosse è altamente preferibile rispetto a tutte le altre. Questo permette di adottare tale sequenza senza valutare alternative, risparmiando così un certo numero di passi di computazione. La macro più facilmente comprensibile, seppure dall'utilità altamente contestuale al livello in cui viene usata, è quella relativa ai **tunnel**. Se Sokoban imbocca un tunnel spingendo una cassa, l'unica scelta che abbia senso è continuare a spostare la cassa e se stesso fino all'uscita del tunnel stesso, rendendo improduttivo il calcolo di decisioni alternative durante la permanenza nel tunnel.

Quello di **corral** (recinto) è un concetto molto importante per la deadlock detection. Si tratta di situazioni in cui la disposizione delle casse abbia reso porzioni del livello inaccessibili a Sokoban: in essenza recinti di casse, le quali possono soltanto, naturalmente, essere spinte verso l'interno del recinto stesso. I corral non garantiscono la presenza di un deadlock, ma costituiscono situazioni interessanti da considerare, poiché possono molto spesso condurre a deadlock. Si consideri che, se si vuole risolvere il livello, i corral vanno necessariamente "sciolti" presto o tardi. Per questa ragione, è opportuno affrontarli appena li si scova, vista l'alta probabilità che possano condurre a deadlock, e dunque a fare il prima possibile pruning dell'albero di ricerca. Gli approcci possibili per esaminare un corral e verificare se è possibile interrompere alcuni rami di ricerca sono numerosi e spesso molto complessi, tanto che il corral pruning può essere considerato come un secondo problema di ricerca parallelo alla risoluzione del livello.

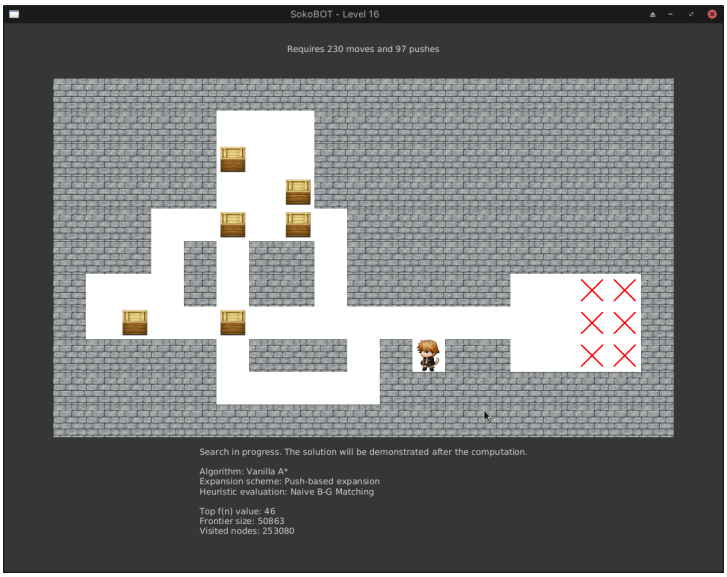
4.8 – Note conclusive sullo sviluppo di SokoBOT

Nell'esaminare l'implementazione concreta di SokoBOT, si noti che l'obiettivo principale erano velocità e chiarezza in fase di prototipazione. È palese che si possano raggiungere prestazioni migliori con una gestione più minuziosa degli algoritmi e delle strutture dati usate, nonché isolando parti parallelizzabili del risolutore. Java stesso non sarebbe nemmeno stato una scelta particolarmente centrata in quel contesto. Ma la questione non ha, dopotutto, particolare importanza. Il risolutore proposto vuole essere un piccolo specchio indicativo delle tecniche di risoluzione per Sokoban, non un mostro di number crunching da mandare in esecuzione sui supercomputer del MIT per infrangere qualche record. In virtù di ciò, evito di produrmi in particolari raccolte di dati o plotting statistici su tempi di esecuzione e uso di memoria, perché sarebbe un po' come fare esperimenti di fisica nucleare usando una grossa fionda invece che un acceleratore di particelle.

Per quanto riguarda i dettagli strettamente implementativi – nonché le istruzioni per poter provare SokoBOT -, invito ancora una volta a consultare il repo GitHub. Non ho prodotto documentazione particolarmente formale, ma i commenti dovrebbero bastare a comprendere le porzioni più significative dell'implementazione, essendo lo *scope* di questo one-man project, dopotutto, non particolarmente esteso. Ci si accorgerà che ho usato la lingua inglese per readme, messaggi di commit, commenti al codice e testo dell'UI. Sospettavo che l'intersezione tra persone che troverebbero utile il contenuto del repo e persone con alto contenuto di pomodoro di San Marzano nel sangue potesse avvicinarsi pericolosamente all'insieme vuoto. Concludo la relazione con una breve presentazione dell'interfaccia utente.



Menù di selezione dei livelli e configurazione del solver



Schermata di ricerca della soluzione

La semplice GUI inclusa permette di testare tutti i meccanismi risolutivi descritti dalla qui presente relazione su una selezione di 17 livelli. Per quanto riguarda la deadlock detection, le varie strategie sono selezionabili in isolamento, azionate contemporaneamente oppure disattivate completamente. Inoltre, la scelta del criterio di stima euristica va intesa *dove applicabile.* La BFS, nella fattispecie, non fa uso di euristiche.

Una volta avviata la ricerca di una soluzione, l'interfaccia mostrerà dati rilevanti sulla ricerca in corso. A ricerca ultimata, la soluzione trovata sarà mostrata dinamicamente su una board di gioco. Si possono, peraltro, prendere le redini in mano e giocare manualmente i livelli. Tale possibilità è stata introdotta principalmente a scopo di debugging e non si tratta di certo di un'esperienza di gioco particolarmente raffinata, ma è pienamente usabile.