ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

Seconda Facoltà di Ingegneria
con Sede a Cesena
Corso di Laurea Specialistica in Ingegneria Informatica

TICKET TO RIDE

Elaborata nel corso di: Linguaggi e Modelli Computazionali
LS

Relazione di Progetto di:
DENIS BRIGHI, LUCA DOMENICONI

Relatore:
Prof. MIRKO VIROLI

ANNO ACCADEMICO 2010–2011
SESSIONE III

PAROLE CHIAVE

Ticket To Ride

ABM - Agent Based Modeling

Artificial Intelligence

Board Game

Prolog

Indice

Struttura della relazione

Questa relazione sarà così strutturata: nel primo capitolo verrà introdotto il gioco, attraverso una sua definizione struttura le e di interazione tra i giocatori; nel secondo capitolo è descritta in breve la progettazione del nostro sistema, mentre nel terzo capitolo è esplicitata la parte relativa all'intelligenza artificiale dei giocatori virtuali.

Capitolo 1

Spiegazione del gioco

Ticket to ride è un gioco da tavolo di stile tedesco[1] di ambientazione ferroviaria che permette a 2 (fino a 5) giocatori di sfidarsi nel costruire una rete ferroviaria, collegando le città presenti sul tabellone secondo degli obiettivi forniti dalle carte.

1.1 Descrizione del Problema

Regole del gioco:

I giocatori dispongono di 45 treni dello stesso colore,5 carte vagone ferroviario coperte e 3 carte destinazione. All'inizio del gioco, i giocatori scelgono almeno una carta destinazione tra quelle fornite. Queste carte mostrano una coppia di città presenti sulla mappa di gioco, esse sono gli obiettivi: rappresentano i due estremi che i giocatori devono cercare segretamente di collegare. Ogni obiettivo dipendentemente della lunghezza della tratta da collegare conferisce un punteggio che verrà aggiunto al termine del gioco al punteggio complessivo del

[1] I giochi da tavolo in stile tedesco è un'ampia classe di giochi caratterizzati generalemente da semplici regole, una durata medio-corta, interazioni indiretta tra giocatori e componenti fisica interessante. Il gioco enfatizza la strategia, minimizzando il fattore fortuna e conflittuale, con una forte propensione per l'economia piuttosto che temi militari, di solito mantenendo tutti i giocatori attivi fino al termine della sessione. I giochi da tavola in stile tedesco sono spesso in competizione con quelli in stile americano, in quanto questi ultimi risaltano il fattore fortuna e conflittualità tra i player.

giocatore, se la coppia di città è stata effettivamente collegata tramite i suoi treni. Al contrario il punteggio verrà scalato, nel caso il collegamento non sia stato effettuato con successo. Per raggiungere gli obiettivi, si devono utilizzare i cosiddetti itinerari cioè dei collegamenti tra città composte da uno o più vagoni (segmenti) dello stesso colore. Ognuno di questi può essere occupato da uno ed un solo giocatore. Per completare un itinerario, occorre scartare un numero di carte vagone ferroviario pari al numero di segmenti che compongono l'itinerario stesso, che abbiano tutte lo stesso colore dei segmenti. Una volta scartate le carte basterà occupare l'itinerario con un numero di propri treni pari al numero di segmenti che compongono l'itinerario stesso. Per ogni itinerario occupato, il giocatore guadagna dei punti, dipendentemente dalla lunghezza dell'itinerario stesso:

1. 1 segmento - 1 punto
2. 2 segmenti - 2 punti
3. 3 segmenti - 4 punti
4. 4 segmenti - 7 punti
5. 5 segmenti - 10 punti
6. 6 segmenti - 15 punti

Ad esempio le città Portland e San Francisco, sono collegate tramite un itinerario composto da 6 segmenti di colore verde. Per poterlo completare occorrerà scartare 6 carte vagone ferroviario, tutte rigorosamente verdi e porre 6 trenini del proprio colore sull'itinerario. Il guadagno di punti da parte del giocatore sarà pari, in questo caso, a 15. Esistono anche delle tratte formate da segmenti di colore grigio (itinerario any). Queste ultime possono essere completate scartando un numero di carte vagone ferroviario di qualsiasi colore, pari al numero di segmenti che compongono la tratta. E' importante che tutte le carte scartare però siano dello stesso colore. Ad esempio la tratta Vancouver - Calgary è formata da 3 segmenti grigi e quindi può essere completata scartando 3 carte blu oppure 3 carte verdi etc, etc... Oltre alle carte vagone ferroviario colorate, esiste anche un'altro tipo di carta detta jolly, che si distingue dalle altre perché presenta una

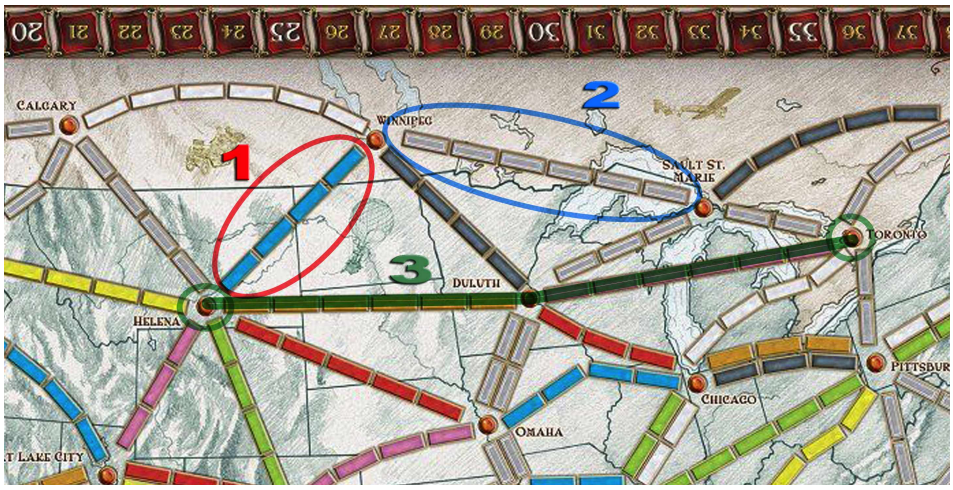

Figura 1.1: Zoom del tabellone: 1)Itinerario Colorato, 2)Itinerario Any 3)Tratta Toronto-Helena

locomotiva disegnata sul fronte invece che un vagone. Quest'ultima può essere usata nella fase di scarto, attribuendogli il colore voluto (come se fosse appunto un jolly).

Sul tavolo saranno presenti:

1. La mappa che rappresenta la rete ferroviaria degli Stati Uniti, dove sono visibili le varie città da collegare con le diverse tratte colorate.

2. Una pila di carte vagone ferroviario (carte colore) e/o jolly coperte, denominata mazzo coperto.

3. Una lista di 5 carte vagone ferroviario (carte colore) e/o jolly scoperte, denominata mazzo scoperto.

4. Una pila di carte obiettivo (carte destinazione supplementari) denominata, mazzo tratte.

Ad ogni turno, i giocatori possono:

- raccogliere due carte dal mazzo coperto

- raccogliere due carte dal mazzo scoperto
- raccogliere una carta dal mazzo coperto ed una dal mazzo scoperto
- raccogliere un jolly dal mazzo scoperto
- raccogliere 3 carte obiettivo dal mazzo tratte e scartarne al massimo 2
- scartare le carte vagone ferroviario (e/o jolly) già raccolte per completare una tratta libera

Il gioco termina quando un giocatore rimane con un numero di treni inferiore a 3 ancora da giocare. Quando si verifica ciò, ogni giocatore gioca un ultimo turno, dopo di che ciascuno rivela le sue carte destinazione nascoste e si passa al calcolo dei punteggi. Vince naturalmente chi ha raggiunto il punteggio maggiore, sulla base di tutti i punti guadagnati scartando le carte vagone, sommati ad i punti per le tratte completate (o chiaramente sottratte nel caso non siano state portate a termine).

1.2 Analisi del Problema

Caratteristiche: gioco da 2 a 5 giocatori (di cui uno ed uno solo necessariamente umano)

Fase di configurazione (prima dell'inizio della partita ogni giocatore dovrà avere) :

- 5 carte vagone
- 45 trenini
- 3 carte tratte da scegliere. Possono esserne scartate al massimo 2.

Inoltre vengono scoperte 5 carte vagone in modo tale da creare il mazzo scoperto. In questa fase analizziamo quello che è il ciclo che compone una partita.

Partenza della partita. Turno del giocatore X, il giocatore può:

- prendere 2 carte vagone dal mazzo scoperto o dal mazzo coperto
- perdere 1 carta vagone dalle carte scoperte e una dal mazzo coperto (o viceversa)
- prendere 1 jolly dal mazzo scoperto
- prendere 3 carte tratte e scartarne 2 al massimo
- scartare X carte dello stesso colore per riempire un itinerario di X trenini che non sia già occupato da un altro giocatore

Scelta una di queste azioni vengono controllati il numero di trenini del giocatore X. Se questo numero è inferiore a 3, il prossimo giocatore inizierà l'ultimo turno che finirà con il giocatore X compreso.

Dopo questo controllo si passa al giocatore successivo.

Terminato l'ultimo turno vengono conteggiate le tratte completate/non completate per ogni giocatore ed aggiunti/tolti i punti relativi

1.3 Analisi della struttura

1.3.1 Entità che compongono il sistema

Andiamo ora ad analizzare le entità viste nel nostro problema, che risultano essere:

1. giocatori: entità che contengono il comportamento dei giocatori e tutte le informazioni riguardanti la corretta pianificazione delle loro scelte
2. il mazzo coperto : entità che mappa il mazzo coperto, in modo tale da fornire le funzionalità per interagirvi
3. il mazzo scoperto : idem come sopra
4. il mazzo tratte : idem come sopra
5. il tabellone : entità che mappa il tabellone. Esso deve mantenere tutte le informazioni riguardanti la partita in corso, come

i punteggi, lo stato dei giocatori (informazioni sui trenini rimanenti e le carte in mano) e lo stato dei vari itinerari, oltre che chiaramente fornire tutte le funzionalità per interagirvi.

A meno che i giocatori abbiano un intelligenza tale da saper gestire i turni autonomamente (in una sorta di controllo decentralizzato, come accade nella realtà) occorre che qualche altra entità gestisca in maniera centralizzata questo compito.

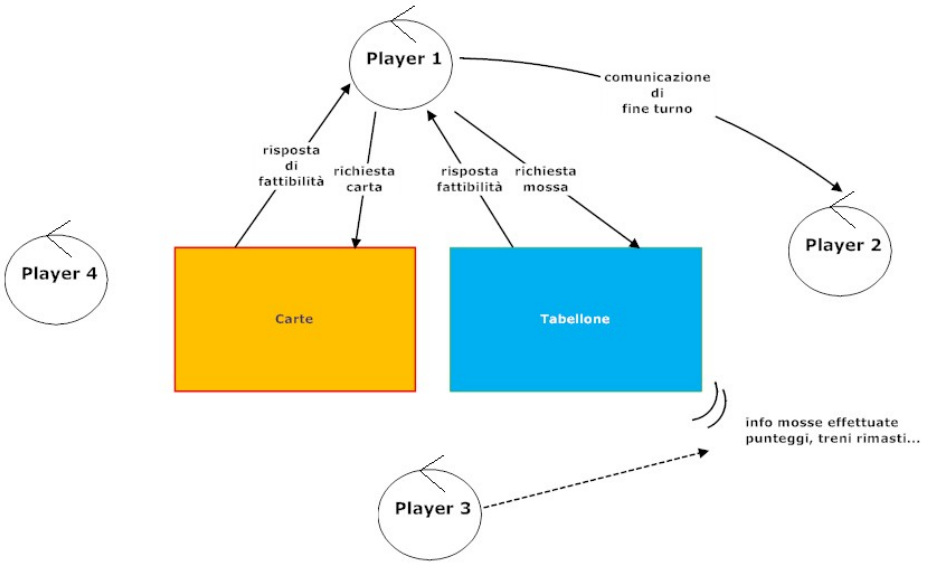

Figura 1.2: Struttura con controllo decentralizzato

Quindi deve essere presente un'entità che si preoccupa di interagire con le altre varie entità viste sopra, per realizzare quella che è la logica del gioco. In particolare occorre che essa faccia giocare i vari player nel momento giusto, preoccupandosi quindi di tutta la gestione dei turni fino alla fine della partita. Per semplicità, preferiamo focalizzarci su un controllo centralizzato visto che costruire un approccio decentralizzato sarebbe sicuramente più dispendioso in termini di tempo e non darebbe vantaggi significativi al nostro sistema. Inoltre dato che il tabellone contiene tutte le informazioni sulla partita in corso, sarà il tabellone l'entità che si preoccuperà di portare a termine il compito di gestire la partita.

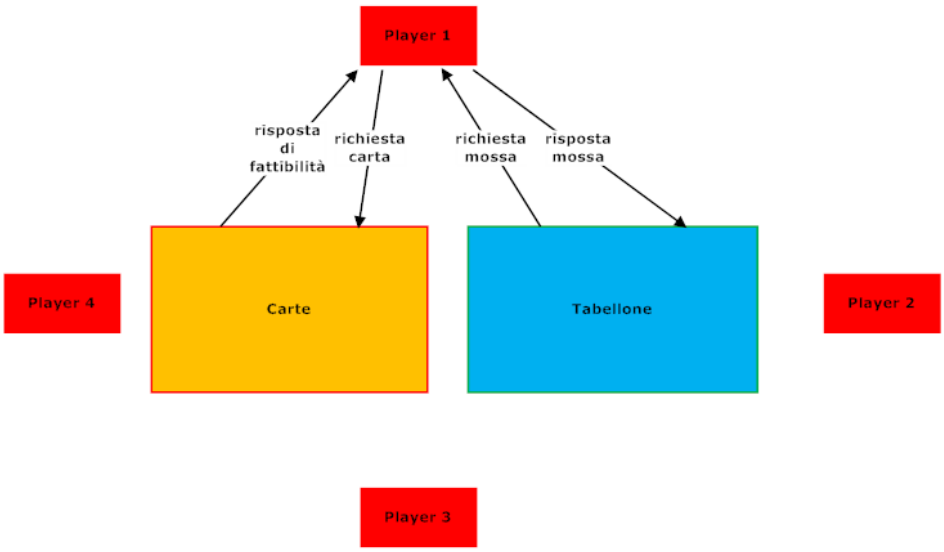

Figura 1.3: Struttura con controllo centralizzato

Dato che il controllo sarà centralizzato, mappiamo tutte le entità su degli oggetti, così come i giocatori che non avranno la necessità di essere agenti visto il tipo di struttura scelta.

1.3.2 Funzionalità delle entità

Andiamo ad analizzare ora quelle che dovranno essere le funzionalità di interfacciamento delle nostre entità:

- giocatori :
  - esegui il turno
- mazzo coperto :
  - restituisci una carta
  - reimmetti nel mazzo una carta
- mazzo scoperto :
  - mostra la lista di carte scoperte

  - restituisci una determinata carta scelta, togliendola dal mazzo scoperto
  - inserisci una carta nel mazzo
  - togli tutte le carte dal mazzo se sono presenti più di 3 carte jolly e inserisci tutte le carte nuove

- mazzo tratte :
  - restituisci una tratta
  - reimmetti nel mazzo una carta

- tabellone :
  - restituisce le informazioni grafo (la tratta X è occupata?)
  - restituisce le informazioni punteggio
  - restituisce trenini rimanenti giocatore n.
  - occupa tratta (numero n) con trenini colore (colore x) (risposta vero/falso se è possibile occupare o meno la tratta, se è vero aggiorna punteggio e grafo e carte colore giocatore)
  - richiedi carta scoperta (poi preleva carta coperta da mazzo carte coperte e rimpiazza con questa)
  - richiedi l'elenco delle carte scoperte

  Le funzionalità che sono elencate in seguito, sono già fornite da altre entità. Il tabellone però le incapsula, in modo tale da far rispettare le regole del gioco (ad esempio se vengono richieste più di X carte coperte in un turno, il tabellone deve necessariamente impedire questa azione).
  - richiedi carta coperta
  - richiedi carta scoperta
  - richiedi carte tratta
  - accetta carte tratte

Capitolo 2

Progettazione

Come abbiamo visto nel paragrafo precedente, le entità che devono essere modellate sono le seguenti:

1. giocatori: entità che contengono il comportamento dei giocatori e tutte le informazioni riguardanti la corretta pianificazione delle loro scelte

2. il mazzo coperto : entità che mappa il mazzo coperto, in modo tale da fornire le funzionalità per interagirvi

3. il mazzo scoperto : idem come sopra

4. il mazzo tratte : idem come sopra

5. il tabellone : entità che mappa il tabellone. Esso deve mantenere tutte le informazioni riguardanti la partita in corso, come i punteggi, lo stato dei giocatori (informazioni sui trenini rimanenti e le carte in mano) e lo stato dei vari itinerari, oltre che chiaramente fornire tutte le funzionalità per interagirvi. Inoltre si prenderà cura di portare a termine il controllo dei turni, gestendo di fatto tutto il gioco.

Si può subito notare come le entità intelligenti cioè quelle che devono cercare di simulare un ragionamento umano sono i *giocatori*. Per questo motivo viene naturale utilizzare un linguaggio di programmazione adatto a questo tipo di lavoro, come il prolog. Questo significa

anche che tutte le informazioni necessarie per portare a termine un ragionamento efficace devono essere fornite al giocatore in questione.

Le entità come *mazzo scoperto*, *coperto* e *tratte*, devono invece portare a termine delle funzionalità molto semplici , come ad esempio restituire una carta e reimetterla nel mazzo, per questo motivo si è scelto di utilizzare dei semplici oggetti java per mapparle.

Per quanto riguarda la View del nostro sistema, sarà ovviamente utilizzato Java, in quanto esso ci offre le librerie per una semplice e rapida composizione delle nostre viste. Per aumentare le performance e rendere l'implementazione più semplice si è deciso di mantenere tutto il modello (e quindi tutte le informazioni) in java poiché il numero di entità descritte in questo linguaggio che devono accedere alle informazioni contenute nel modello, sono molte di più di quelle scritte in prolog. Nel caso che le entità *giocatori* debbano entrare in possesso di informazioni riguardanti il gioco, sarà necessario passargli parte del modello sotto forma di copia.

Capitolo 3

Ragionamento

Abbiamo deciso di sviluppare l'intelligenza dei giocatori virtuali attraverso una implementazione in Prolog di un tipico ragionamento umano, quello che noi per primi abbiamo sperimentato giocando sul tavolo. L'osservazione, i meccanismi, ... sono stati riportati per quanto possibile all'interno dell'AI dei giocatori.

3.1 la forma della conoscenza

Ogni player, durante il proprio turno, ha accesso alla conoscenza dello stato attuale del gioco. Questo avviene attraverso una rappresentazione Prolog del tavolo, sia per quanto riguarda lo spazio condiviso da tutti (quindi mappa, tratte libere/occupate e mazzo scoperto), sia del proprio mazzo carte posseduto.

Abbiamo dunque deciso di rappresentare i vari componenti come segue:

- *cardColor(Id,Colore,Rgb).* Con questo predicato elenchiamo quali sono le tipologie di carte che fanno parte del gioco, associando ad ogni colore sotto forma di stringa –Colore–, un identificativo univoco ed un codice rappresentate un colore in forma RGB. (Esempio: cardColor(2,green,'00ff00').);

- *trainColor(Id, Colore, Rgb).* Attraverso queste dichiarazioni possiamo listare i possibili giocatori, assegnandogli ad ognuno,

analogamente al caso precedente, un identificativo, un colore-pedina ed un relativo codice RGB. (Esempio: rainColor(1,grey,'9c9994').);

- *city(Id, City, X, Y).* L'elenco delle città viene presentato al giocatore mediante l'uso del predicato city. Ogni città è quindi definita da un identificativo, una stringa che ne rappresenta il nome, e due interi che la collocano spazialmente sulla mappa (coordinate rispettivamente x e y). (Esempio: city(3,'Seattle',119,148).);

- *route(Id, IdCityA, IdCityB, Ntrains, IdColor).* È il "link" tra due city, e rappresenta dunque un itinerario tra due città. Ogni route è descritta da un identificativo, le due città che collega, il numero di treni necessari ad occupare l'itinerario e l'identificativo del colore che lo caratterizza (Esempio: route(10,9,8,3,7).);

- *myId(Id).* Identifica il giocatore all'interno della teoria prolog (Es. myId(2).);

- *routeOwner(RouteId, PlayerId).* Ogni qual volta un giocatore occupa un itinerario sarà inserito questo predicato per specificare l'appartenenza della route ad un giocatore (Es. routeOwner(54,1).);

- *colorCard(ColorId).* È la lista delle carte colore in mano al player. Ogni entry rappresenta una carta, e il colore viene rappresentato mediante il suo identificativo numerico (Esempio: colorCard(5).);

- *routeCard(CityA, CityB, Score).* Ogni entry rappresenta un obbiettivo del giocatore: ciascuna carta è infatti rappresentata da una città di partenza, una di destinazione e un punteggio. Il punteggio è aggiunto al giocatore soltanto se alla fine del gioco ci sarà una tratta che li collega (Esempio routeCard(7, 18, 14));

- *visibleCard(ColorId).* Questi predicati rappresentano ciascuna carta visibile sul tavolo da gioco, e ne viene caratterizzato il colore mediante apposito identificativo (Esempio visibleCard(5).);

- *cardTaked.* Questo predicato viene aggiunto alla teoria quando il giocatore pesca, sia che questa appartenga al mazzo visibile

che a quello coperto. Questo ci permette di modellare la regola che durante il turno, se viene pescata una carta, l'unica altra mossa possibile sia quella di pescarne un'altra.

3.2 La strategia

Nell'ambito di un gioco da tavolo, spesso non esiste una strategia ottimale a livello globale. È stato dunque sviluppato un algoritmo euristico basandoci sulla strategia più comune adottata dai player di questo specifico gioco. In seguito abbiamo proceduto ad ottimizzare questa strategia per avere un livello di gioco accettabile, per poter mettere alla prova il giocatore umano.

Possiamo dunque identificare gli obbiettivi della strategia, qui elencati in ordine di priorità:

1. *Il raggiungimento del maggior numero di obbiettivi possibile;*
2. *Il completamento di un obbiettivo;*
3. *Lo scarto di un itinerario favorevole al raggiungimento di un obbiettivo;*
4. *La collezione di carte colore utili al conseguimento di un itinerario colorato ;*
5. *La collezione di carte colore utili al conseguimento di un itinerario "any";*

3.3 La scelta del cammino da percorrere

Allo scopo di raggiungere il numero maggiore di obbiettivi dunque è naturale cercare di raggiungere, e quindi di collezionare carte, nel più breve tratto possibile. In questo contesto l'algoritmo di Dijkstra è lo strumento ottimale, anche se computazionalmente costoso, per il raggiungimento del nostro scopo. Abbiamo dunque implementato l'algoritmo nel seguente modo:

1. *Inizializzazione.*

- Poniamo S=1, T=2,3,...,n, f(1)=0, J(1)=0.
- Poniamo f(i)=p(1,i), J(i)=1 per tutti i nodi adiacenti ad 1.
- Poniamo f(i)=∞, per tutti gli altri nodi.

2. *Assegnazione etichetta permanente*

   - Se f(i)=∞ per ogni i in T STOP.
   - Troviamo j in T tale che f(j)=min f(i) con i appartenente a T:
   - Poniamo T=Tj e S=S$\cup$j
   - Se T=Ø STOP

3. *Assegnazione etichetta provvisoria*

   - Per ogni i in T, adiacente a j e tale che f(i)>f(j)+p(j,i) poniamo:
   - f(i)=f(j)+p(j,i)
   - J(i)=j

4. *Andiamo al passo 2*

Questo algoritmo è stato implementato in prolog attraverso il predicato dijkstra(+CityA, +CityB, -Sol, -Val, -Sol2) ed altri predicati di supporto all'algoritmo.

Listing 3.1: Esempio di utilizzo del predicato dijkstra

```
?-dijkstra(1,36,Sol,Val,Sol2).
yes.
Sol / [1,3,7,16,36] ,  Val / 16,
Sol2 / [['Vancouver',1,1,'Seattle'],['Seattle',4,6,'Helena'],
  ['Helena',9,5,'Omaha'],['Omaha',8,4,'Chicago']]).
```

3.3.1 Route e Dist

Da notare sono i due predicati route1 e dist qui riportati:

Listing 3.2: Predicato route1

```
route1(X, IDA, IDB, NT, COL):-route(X, IDB, IDA, NT, COL).
route1(X, IDA, IDB, NT, COL):-route(X, IDA, IDB, NT, COL).
```

Listing 3.3: Predicato dist

```
dist(IDA,IDB,0):-route1(ID,IDA,IDB,_,_),myId(X),routeOwner(ID,X).
dist(IDA,IDB,NTrains) :- route1(ID,IDA,IDB,NTrains,_),
    not (routeOwner(ID,_)).
```

che sono di fondamentale importanza per l'algoritmo di dijkstra. Il primo infatti permette di invertire il predicato "route", cioè di ricercare un itinerario tra due città in qualsiasi ordine (ad esempio sia da A verso B che da B verso A) senza introdurre maggiore codice di controllo all'interno di altri predicati; "dist" a sua volta è il predicato utilizzato per la ricerca del costo effettivo di un itinerario. Ma vediamone con attenzione il suo funzionamento: se esiste un itinerario da A verso B (o viceversa, dato l'uso di route1), ed esiste un predicato routeOwner per cui il giocatore possiede quella tratta, la distanza fra le due città sarà 0. In caso contrario, possono esserci due alternative: se nessun altro giocatore possiede la tratta, NTrains corrisponderà alla distanza effettiva data dal predicato route; altrimenti risponderà "no" se la tratta è già stata occupata. Questo predicato è molto utile in quanto "elimina" automaticamente le tratte già occupate da altri giocatori e quindi non più disponibili, ed inoltre permette all'algoritmo di dijkstra di percorrere gli itinerari in possesso a "costo zero". Questo fatto torna utile in più di una occasione:

- Durante lo svolgimento del normale gioco, una volta occupata un itinerario, non sarà più necessario collezionare carte di quel tipo.

- Avere itinerari a costo zero, comporta anche una modifica alla strategia globale, in quanto per altri obbiettivi sarà possibile inserire in soluzione quell'arco senza aggiungere ulteriori costi, e magari trovare percorsi alternativi a costo totale minore.

- Il controllo a fine gioco del raggiungimento degli obbiettivi è semplificato, in quanto basta controllare che la soluzione dell'algoritmo di dijkstra tra le due città sia zero, qualsiasi percorso sia stato realizzato.

A sua volta, l'inibizione di archi del grafo è un'ottima strategia per evitare di mettere in soluzione archi oramai occupati da altri giocatori.

3.3.2 Creazione di un piano di gioco

Il vero punto a favore della nostra strategia è la creazione di un piano di gioco, cioè la creazione di un insieme di liste contenenti le città da toccare dal nostro percorso alla ricerca degli obbiettivi. Il giocatore virtuale eseguirà infatti ad ogni turno queste operazioni:

1. Piano:
    - Controlla se hai un piano; se si controlla che sia ancora valido;
    - Se non hai un piano, crealo;
2. Seleziona la prossima mossa da eseguire;

Ma andiamo più nel dettaglio descrivendo come procede effettivamente ciascun player virtuale. L'agente ha un proprio stato interno, definito dal piano di gioco che esso vuole attuare. Questo non è altro che l'insieme delle tratte che vorrebbe percorrere. All'inizio del gioco il piano è ovviamente vuoto, per cui ciascuna AI andrà a compilarlo. A questo proposito il predicato plan(X) restituirà, per ogni routeCard (estrapolando dunque CityA e CityB) il percorso minimo dato dall'algoritmo di dijkstra. Il predicato intermedio paths introduce il controllo nei casi in cui dijkstra ritorni una soluzione a costo zero, e quindi è possibile eliminarla dal piano, oppure nessuna soluzione, quando non esiste nessun percorso tra le due città.

Listing 3.4: Predicato plan(X)

```
plan(L):- findall([A,B],routeCard(A,B),L2), paths(L2, L1),
2                 quicksort(L1, L).

paths([], []).
paths([[A,B]|T], O):-dijkstra(A,B,Sol,Val,_),!,
    (Val=0, !, O=O1;O=[[Val, Sol]|O1]), paths(T, O1).
7
paths([[A,B]|T], O):-retract(start(_)), retract(stop(_)), paths(T, O).
```

Listing 3.5: Esempio di piano

```
?-plan(X).
2 yes.
X / [[20,[32,26,36,16,7,3]],
  [20,[9,8,11,17,23,27,28,29]],
  [17,[24,15,16,11,10,6]]].
```

Come possiamo vedere dall'esempio un sottopiano è formato da un costo ed una lista di città da visitare per raggiungere l'obbiettivo. Infine possiamo definire un piano come una lista di sottopiani. Inoltre plan prevede un ordinamento (quicksort(L1,L)) in base alla lunghezza del percorso decrescente. L'ordinamento crescente ha principalmente due obbiettivi: in primo luogo tratte più lunghe assicurano un punteggio maggiore; in secondo luogo sono sono più difficili da realizzare, e quindi è bene che siano soddisfatte per prime.

3.3.3 Controllo di un piano già creato

Se un player ha a disposizione un piano già preparato nel turno precedente, questo va ricontrollato in quanto gli altri giocatori, durante i loro rispettivi turni, potrebbero aver occupato degli itinerari a cui il giocatore attuale aspira. Il predicato che realizza ciò è il seguente:

Listing 3.6: Predicato checkPlan

```
checkPlan([]).
checkPlan([[_,Itinerario]|T]) :- checkPlan2(Itinerario), checkPlan(T), !.
checkPlan2([H1]).
checkPlan2([H1,H2|T]) :- dist(H1,H2,A), !, checkPlan2([H2|T]), !.
```

Per ogni itinerario, si va a controllare che esista una distanza (può essere anche 0!) attraverso il predicato dist: se questo non fallisce per ogni coppia di città H1, H2 all'interno del piano allora il piano è ancora valido ed è possibile procedere alla selezione della prossima mossa da eseguire.

La creazione di un piano e il controllo della correttezza ci permette di utilizzare agevolmente l'algoritmo di dijkstra solo nei casi in cui è STRETTAMENTE necessario, riducendo così le risorse computazionali richieste.

3.3.4 Selezione della prossima mossa

Una volta che un giocatore possiede un piano valido ed avendo le informazioni riguardanti le carte dei mazzi e quelle in suo possesso, può iniziare ad applicare una strategia per il raggiungimento dei suoi obbiettivi. Tra le varie opzioni che il giocatore può scegliere, abbiamo dato una priorità ad ognuna come segue:

1. (scelta 0 - presa di carte obbiettivo)
2. occupazione di una tratta “Colorata”
3. occupazione di una tratta “Any”
4. presa di carte dal mazzo scoperto
5. presa di un jolly dal mazzo scoperto
6. presa di carte dal mazzo coperto

Questa rimane comunque solo una tipologia di strategia, nulla vieta di realizzarne delle altre. Per questo motivo, si è sviluppata la strategia come modulo a parte all'interno del programma, in modo da cambiarla agevolmente senza bisogno di integrazioni aggiuntive col resto del software.

Listing 3.7: Selezione della prossima mossa in Prolog - plan01.pl

```
selectMove(takeRouteCards):- not(cardTaked), plan([]), !.

selectMove(O):- not(cardTaked), plan(X), tryDiscardAll(X, O), !.

selectMove(O):- plan(X), lookForVisible(X, O), !.

selectMove(O):- plan(X), checkAny(X), nonInterestingCards(X, RC),
    handColorCounter(RC,Out), quicksort(Out,O1),
    selectAVisibleCard(O1,O), !.

selectMove(takeVisible(1)):-not(cardTaked), visibleCard(1).

selectMove(takeHidden).
```

Vediamo più in dettaglio nelle prossime sezioni le strategie per ogni mossa.

3.3.4.1 Occupare un itinerario “Colorato” o “Any”

Com'è facile capire, quando si hanno in mano carte di colore tale per cui è possibile occupare un itinerario a cui si è interessati (cioè presente nel piano) è bene farlo, prima che lo facciano altri giocatori. Per questo motivo questo mossa ha avuto la priorità maggiore tra le possibili scelte dell'agente. L'occupazione delle tratte implica però un'altro fattore di priorità: l'occupazione di itinerari colorati infatti richiede la collezione di carte dello stesso specifico colore, mentre quelle per le

tratte di colore grigio è possibile utilizzare qualsiasi tipo di carta. Per questo motivo abbiamo dato priorità maggiore agli itinerari di colore fisso perché di maggiore difficoltà.

Listing 3.8: Occupazione delle tratte

```
tryDiscardAll([[_,Tratta]|T], X):- tryDiscard(Tratta, X), !.
tryDiscardAll([[_,_]|T], O):- tryDiscardAll(T, O).

tryDiscard([H1,H2|T], X):-dist(H1, H2, P), P < 1, !,
     tryDiscard([H2|T], X).

tryDiscard([H1,H2|T], takeRoute(IDR, Col, NT, 0)):-
  route1(IDR, H1, H2, NT, Col), Col>1, findall(Col, colorCard(Col), L),
  length(L, N), N>=NT.

tryDiscard([H1,H2|T], takeRoute(IDR, Col, N, NUJ)):-
    route1(IDR, H1, H2, NT, Col), Col>1,
    findall(Col, colorCard(Col), L), findall(1, colorCard(1), LJ),
    length(L, N), length(LJ, NJ), TOT is N+NJ, TOT>=NT, NUJ is NT - N.

tryDiscard([H1,H2|T], takeRoute(IDR, COLORE, NT, 0)):-
    route1(IDR, H1, H2, NT, 1), plan(X),  nonInterestingCards(X,O),
    handColorCounter(O,O1), member([NT, COLORE], O1),!.

tryDiscard([H1,H2|T], takeRoute(IDR, COLORE, NT, 0)):-
    route1(IDR, H1, H2, NT, 1), plan(X), nonInterestingCards(X,O),
    handColorCounter(O,O1), quicksort(O1,O2), member([NT1, COLORE], O2),
    NT1>=NT.

tryDiscard([_,H2|T], O):- tryDiscard([H2|T], O).
```

Il predicato tryDiscardAll(+X, -O), dato un piano X, resistituisce un fatto takeRoute(IdR, Col, N, NJ), dove IdR è l'id della route da occupare, Col è l'identificativo del colore delle carte, N è il numero di carte colore utilizzato, NJ è il numero di jolly utilizzato; N+NJ deve essere sempre uguale a NT dato da route1(IDR, H1, H2, NT, Col), cioè il numero dei treni necessari per occupare tale itinerario. Il razionale che sta dietro a questo predicato è molto semplice: per per ogni coppia di città (H1, H2) il cui colore non sia il grigio, si cerca nel proprio mazzo se si dispone di sufficienti carte per la sua occupazione (N≥NT), senza o con l'utilizzo di Jolly se posseduti (N+NJ≥NT). In caso contrario, si procede selezionando dalle proprie carte quelle che non sono "destinate" a itinerari colorati, ed utilizzandole per occupare delle tratte grigie. Anche qui il ragionamento alla base è semplice: tutte le carte che servono per occupare tratte colorate vengono messe da parte e le restati sono utilizzate per occupare tratte any, ottimizzando così la strategia. Se nessuna delle scelte è applicabile, si passa alla prossima

coppia di città. Com'è possibile notare, per l'occupazione di tratte any non è stato implementato l'uso di jolly; si è inoltre preferito scegliere, se possibile, tra tutti i colori delle carte "non interessanti" che il giocatore ha in mano, quello il cui numero di carte è strettamente uguale al numero di treni necessari. Questo fa in modo che vengano utilizzate il numero minore possibile di carte per occupare una tratta any. Ad esempio, se nonInterestingCards restituisce che i colori non necessari sono (6 rosa, 5 carte di colore blu, 3 gialle e 2 verdi) e abbiamo una tratta any di lunghezza 2 verrà scelto il verde, mentre se la tratta fosse di lunghezza 4 il nostro algoritmo sceglierebbe il rosa.

Listing 3.9: Predicati utili alla strategia: nonInterestingCards

```
nonInterestingCards(X, RC):- findall(P, colorCard(P), HC),
  findall(F, visibleCard(F), VC), deleteInterestingCards(X, HC, RC).

deleteInterestingCards([], RC, RC).
deleteInterestingCards([[_,Tratta]|T], HC, RC):-
  deleteInterestingCard(Tratta, HC, RC1), !,
  deleteInterestingCards(T, RC1, RC).

deleteInterestingCard([H], RC, RC):-!.

deleteInterestingCard([H1,H2|T], HC, RC) :- dist(H1, H2, P), P < 1, !,
  deleteInterestingCard([H2|T], HC, RC).

deleteInterestingCard([H1,H2|T], HC, RC) :- route1(_, H1, H2, NT, Col),
  !, member(HC, Col, [], RC1), deleteInterestingCard([H2|T], RC1, RC).
```

Listing 3.10: Predicati utili alla strategia: handColorCounter

```
handColorCounter([H|T],[[N,H]|Rest]) :- findall(H,member(H,[H|T]),L),
  length(L,N), member([H|T],H,[],O),!, handColorCounter(O,Rest).

handColorCounter([],[]).
```

3.3.4.2 Prendere carte colore dal mazzo scoperto

Scelta successiva all'occupazione delle tratte è la presa delle carte dai mazzi. Avendo identificato come "rischioso" la scelta di prelevare una carta dal mazzo coperto, e dunque relegandola come ultima opzione possibile, è chiaro come la collezione di carte "interesting" sia la prima scelta da percorrere. Il predicato lookForVisible(+X, -O), dato il piano X, se ha successo restituisce O=takeVisible(Col), dove il colore sarà uno di quelli che serviranno per completare un itinerario colorato.

Listing 3.11: Predicato lookForVisible

```
lookForVisible([[_,Tratta]|T], takeVisible(X)):-
  findVisible(Tratta, X), !.

lookForVisible([_|T], X):- lookForVisible(T, X).

findVisible([H1,H2|T], X) :- dist(H1, H2, P), P < 1,
  !, findVisible([H2|T], X).

findVisible([H1,H2|T], Col) :- route1(_, H1, H2, NT, Col),
  Col > 1, visibleCard(Col), !.

findVisible([_,H2|T], X) :- findVisible([H2|T], X).
```

Se non ci sono a disposizione carte interessanti, passiamo a collezionare carte di cui già disponiamo almeno una carta nel nostro mazzo. Il razionale è quello di accumulare carte che sono già in nostro possesso, aumentando la probabilità al prossimo turno di scartare tratte any.

Listing 3.12: Predicato per la scelta di una carta utile per itinerari any

```
selectMove(O):- plan(X), checkAny(X), nonInterestingCards(X, RC),
    handColorCounter(RC,Out), quicksort(Out,O1),
    selectAVisibleCard(O1,O), !.
```

Il predicato checkAny(X) controlla se esiste almeno una tratta any all'interno del piano. Se non fosse così sarebbe inutile collezionarne ancora (converrebbe a questo punto tentare la fortuna con una carta). Successivamente si ordinano le carte non interessanti per numero ed infine si cerca se ne esiste una di queste tra quelle scoperte sul tavolo, attraverso il predicato selectAVisibleCard.

Listing 3.13: Predicato checkAny e selectAVisibleCard

```
checkAny([[_,Tratta]|T]) :- checkAny2(Tratta), !.
checkAny([[_,Tratta]|T]) :- checkAny(T).
checkAny2([H1,H2|T]) :- route1(_, H1, H2, NT, 1), !.
checkAny2([H1,H2|T]) :- checkAny2([H2|T]).

selectAVisibleCard([[N,Col]|T],takeVisible(Col)) :- visibleCard(Col), !.
selectAVisibleCard([[N,Col]|T],O) :- selectAVisibleCard(T,O).
```

Se nessuna delle ipotesi precedenti è applicabile, e sussistono le seguenti condizioni:

- non si è ancora pescata nessuna carta dai mazzi
- è disponibile un Jolly nel mazzo scoperto

allora è possibile prelevare un jolly (Colore = 1) dal mazzo scoperto.

Listing 3.14: Presa di una carta Jolly

```
selectMove(takeVisible(1)):-not(cardTaked), visibleCard(1).
```

Come ultimo caso, si procede alla presa di una carta coperta

Listing 3.15: Presa di una carta coperta

```
selectMove(takeHidden).
```

3.3.4.3 Prendere delle tratte Obbiettivo

Se ad un certo punto della partita il piano risulterà vuoto, significherà che abbiamo terminato i nostri obiettivi oppure non è più possibile completarli , poiché ad esempio altri giocatori hanno occupato degli itinerari per noi necessari. A questo punto se abbiamo ancora abbastanza treni a disposizione sarà possibile pescare altre carte tratta per tentare di completare nuovi percorsi, cercando di aumentare in questa maniera il nostro punteggio. Nel caso si decida di pescare nuove tratte, occorrerà anche scegliere tra quelle proposte, scartando quelle che riteniamo non fattibili. Come poter realizzare questa scelta? Il fattore determinante sono ovviamente i treni rimasti ed il costo delle tratte considerate sulla base dello stato attuale del gioco. Il nostro algoritmo prende in ingresso le tre nuove carte obiettivo (quindi una lista di terne: punteggio;città A;città B) ed inoltre il numero di trenini rimasti al giocatore in questione. Come prima cosa, vengono cercati attraverso l'algoritmo di Dijkstra, tutti i costi per raggiungere le città proposte dalle carte, utilizzando il predicato takeIdCostRouteCard(LI,LO).

Listing 3.16: Calcolo dei costi per collegare le città presenti nelle nuove carte tratta

```
takeIdCostRouteCard(LI,LOID) :- takeId(LI,LOID).
takeId([[ID,[A,B]]|T],[[ID,Val] | TO]) :- dijkstra(A,B,Sol,Val,Sol2),!,
    takeId(T,TO).
takeId([[ID,[A,B]]|T],TO) :- takeId(T,TO).
takeId([],[]).
```

La lista in uscita sarà formata dall'insieme di oggetti:

```
ID Carta, Costo attuale per completarla
```

Ad esempio:

```
[[1,20],[2,0],[3,6]]
```

Nel caso una carta contenga due città non raggiungibili, non viene inserito nessun costo e nessun id in lista, mentre se il costo è pari a zero significa che l'obiettivo è già soddisfatto.

Successivamente vengono create le disposizioni di K elementi delle carte tratta, con K che va da 0 a N (con N pari al numero di elementi in lista), attraverso il predicato getAllDisp(LI,LO).

Listing 3.17: Calcolo delle disposizioni delle nuove carte tratta

```
getAllDisp(LI,LO) :- findall(X,getSingleDisp(LI,X),LO).
```

Le disposizioni di 3 carte (con K che va da 0 a 3) di ID pari a:

```
[1,2,3]
```

saranno :

```
[[1],[2],[3],[1,2],[2,3],[1,3],[1,2,3],[]]
```

Ovviamente nella lista in uscita saranno presenti anche i costi per ogni carta tratta. Se ad esempio teniamo in considerazione le carte viste in precedenza, avremo:

```
[[1,20],[2,0],[3,6],[[1,20],[2,0]],[[2,0],[3,6]],[[1,20],
[3,6]],[[1,20],[2,0],[3,6]],[]]
```

Arrivati a questo punto occorrerà calcolare i vari costi per ogni singola disposizione in modo da avere una lista con oggetti del tipo:

```
Lista carte, Costo complessivo per completarle tutte
```

Ad esempio:

```
[[1,20],[2,0],[3,6],[[1,2],20],[[2,3],20],[[1,20],26],[[1,2,3],26],[]]
```

Per fare questo si utilizza il predicato calculateCardsCost.

Listing 3.18: Predicato per il calcolo dei costi complessivi delle disposizioni di tutte le carte tratta

```
calculateCardsCost([H|T],[[CF,H1]|T1]) :- calculateOneDisp(H,H1,0,CF),
    calculateCardsCost(T,T1).
calculateCardsCost([],[]).

calculateOneDisp([[ID,C]|T],[ID|T1],CP,CF) :- CNEW is C+CP,
    calculateOneDisp(T,T1,CNEW,CF).
calculateOneDisp([],[],CF,CF).
```

Successivamente la lista viene ordinata sulla base dei costi, in modo tale da prendere la disposizione che ha costo massimo ma che sia minore del numero di treni posseduti dal giocatore in questione. Questo per massimizzare il guadagno di punti ma sempre in modo tale che

il completamento delle tratte sia fattibile. Questo è possibile grazie al predicato getRightCardByCost.

Listing 3.19: Predicato che restituisce la lista di carte con il guadagno di punti massimo

```
getRightCardByCost([[C,L]|T],COST,L) :- C =< COST,!.
getRightCardByCost([[C,L]|T],COST,O) :- getRightCardByCost(T,COST,O).
```

3.4 Un algoritmo a ottimizzazioni successive

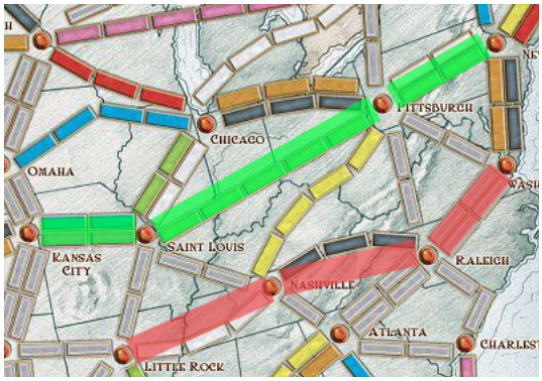

Figura 3.1: Esempio di piano di due tratte obbiettivo: Kansas City-New York(in verde), Little Rock-Washington(in rosso)

Potremmo definire il processo di ricerca del percorso ottimo ad "ottimizzazioni successive", in quanto ogni volta che un giocatore virtuale acquisisce un itinerario, potrà, ricalcolando il piano, ottenere una soluzione uguale o migliore della precedente. Facciamo un esempio. Vediamo in fig.3.1 due obbiettivi: Kansas City-New York e Little Rock-Washington con relativo percorso minimo evidenziato rispettivamente in verde e rosso. Il costo dei percorsi è rispettivamente 9 e 8, cioè la somma dei treni per ciascuna tratta. Ammettiamo che durante il gioco venga occupata dal giocatore l'itinerario St.Luis-Pittsburg: il valore del piano si ridurrà così di 5, dato che la tratta è composta

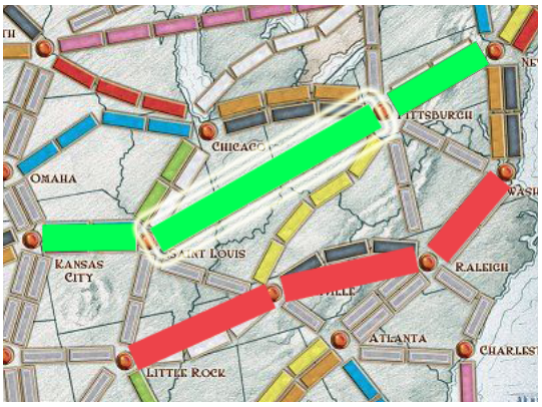

Figura 3.2: Acquisizione dell'itinerario St. Luis-Pittsburg

da 5 trenini. Il punto sostanziale è che, ricalcolando il piano su questo tabellone aggiornato, anche la seconda tratta verrà aggiornata, in quanto il sistema vedrà l'itinerario appena acquisito a costo 0, trovando un nuovo percorso più breve verso l'obbiettivo. Il risultato lo si

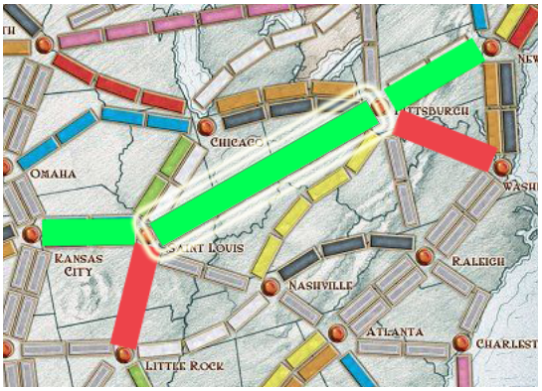

Figura 3.3: Piano strategia aggiornato

può vedere in fig. 3.3 nel quale la tratta rossa è stata modificata. Ora i costi rimanenti delle due tratte sono entrambi 4.

Ovviamente questo è un caso limite in positivo, nel quale si riesce a fruttare un'itinerario, o un insieme di itinerari per più tratte. Il

sistema non riesce a sfruttare a priori l'intersezioni di tratte, ma di volta in volta che si sviluppa la partita trova soluzioni ottime nel dato momento in cui è creato il piano.

Capitolo 4

Integrazione Java-Prolog

Come già accennato nel capitolo precedente, abbiamo voluto realizzare l'intelligenza artificiale dei player virtuali in maniera modulare, così da poter prevedere diverse strategie senza intaccare la parte Java del progetto (a patto di mantenere le signature dei predicati già visti). Per realizzare ciò abbiamo dunque diviso la teoria in 3 parti:

- *dijkstra* è il modulo che contiene l'algoritmo di routing e i predicati al supporto di "plan(X)";
- *checkPlan* è la parte destinata alla verifica del piano;
- *plan01* contiene i predicati atti ad applicare la strategia al piano creato precedentemente. Esso contiene "selectNextMove(O)".

Vediamo ora come è stata realizzata l'integrazione di Prolog all'interno di Java; la classe di riferimento di questo capito è Player.java, la quale contiene il codice sotto riportato. *L'approccio utilizzato è quello di utilizzare il core Prolog come libreria, settandone di volta in volta Teoria ed eseguendo query sui predicati già ampiamente descritti.*

4.1 Player.java e la base dati

Com'è normale pensare in un ambiente Object-Oriented quale è Java, un player virtuale è mappabile in una classe. È facile così realizzare un ambiente multigiocatore semplicemente creando più istanze della stessa classe. Il punto di maggior interesse di Player è il metodo

"move" che ha il compito di decidere la prossima azione da eseguire. Il metodo viene chiamato dall'oggetto Board (che funge da medium di coordinazione tra i vari giocatori), avendo come argomenti la situazione aggiornata del tavolo che lo stesso Board mantiene al suo interno. Possiamo dunque osservare in questo modo: *public void move(Theory dbP, PlayerDeck pd, VisibleDeck vd, RouteDeck rd, int remainingTrains, boolean second) ...*

- *dbP* è un oggetto di tipo Theory che mantiene tutto l'elenco dei predicati che vanno dalla lista dei colori alla definizione del grafo delle città;
- *pd* e *vd* contengono, rispettivamente, le informazioni sull'elenco delle carte in possesso e su quelle del mazzo visibile;
- *rd* è l'elenco degli obbiettivi del giocatore in questione;
- *remainingTrains* rappresenta il numero di treni rimanenti del giocatore;
- *second* informa il player se può effettuare un'ulteriore mossa.

Soffermiamoci un attimo sul parametro "second" per meglio comprendere l'interazione tra Board e Player: la prima entità infatti ha il compito di selezionare uno ad uno i giocatori, farli "pensare" ed eseguire le operazione richieste da loro richieste. Dunque in che senso il giocatore può effettuare "due mosse" ? C'è infatti un solo caso in cui è necessario dover far compiere all'IA due "mini-mosse", quando un player decide di prendere una carta dai mazzi. Ogni qual volta si pesca una carta, lo stato del tavolo o del proprio mazzo cambia, ed è necessario ripetere nuovamente il ragionamento, aggiornando i dati. Attenzione: non è necessario ricalcolare il piano, ma solo la prossima mossa: per questo il parametro second viene settato a Vero da Board quando il player ha la seconda opportunità di pescare un'ulteriore carta.

La base dati è dunque compilata a partire dagli oggetti sopra descritti, i quali a loro volta sono in grado di dare una rappresentazione di loro stessi in formato Prolog, secondo i predicati standard sopra descritti. Ad ogni turno Player controllerà la presenza di un piano o

la sua validità, creandone uno nuovo in caso alla necessità, procedendo infine alla scelta di una mossa.

4.2 Creazione di un piano

Ogni volta che si vuol fare una nuova mossa, occorrerà sapere quali sono i percorsi più corti per collegare i nostri obiettivi, in modo tale da valutare quali carte è meglio pescare oppure se occorre fare altre scelte di gioco. Quindi è necessario creare un cosiddetto piano per decidere come procedere. Se quest'ultimo non esiste, esso dovrà necessariamente creato. Come giù visto nei capitoli precedenti, questa operazione viene effettuata tramite il predicato prolog plan(X), che restituisce una lista contenente i percorsi più brevi per collegare i nostri obiettivi (nel caso non sia possibile collegare nessuna delle città, verrà restituita una lista vuota).

```
if (myPlan==null){
  engine.setTheory(doPlan);
  myPlan = new DbProlog();
  myPlan.append(new Theory(engine.solve("plan(X).")
          .getSolution().toString()+"."));
}
```

4.3 La verifica di un piano

Se il piano è gia stato creato (magari nei turni precedenti) e nel frattempo altri giocatori hanno portato a termine il loro turno, sarà necessario ricalcolarlo poiché ci potrebbero essere stati dei cambiamenti sul tabellone (ad esempio se qualcun altro ha scartato delle carte per occupare degli itinerari che facevano parte del nostro piano). Questo controllo viene fatto ovviamente nella prima parte del nostro turno (cioè quando ancora non è stata scelta alcuna mossa da effettuare). Esso verrà effettuerà tramite il predicato checkPlan(X) già visto nei capitoli precedenti.

```
if (!second) {
  Theory checkPlan = new Theory(belief.toString());
  checkPlan.append(checkPlanTh);
  checkPlan.append(myPlan);
  engine.setTheory(checkPlan);
  if (!engine.solve("plan(X),␣checkPlan(X).").isSuccess()) {
```

```
7       myPlan.append(new Theory(engine.solve("plan(X).")
                .getSolution().toString()+"."));
    }
}
```

Come si può notare, se il checkPlan fornisce una risposta negativa, occorrerà ricreare completamente il piano.

4.4 Selezione della prossima mossa

La selezione della prossima mossa si appoggia all'algoritmo visto nel precedente capitolo, risolvendo il predicato "selectMove(X)." nel seguente modo

```
Theory nextMove = new Theory(belief.toString());
nextMove.append(algorithm);
nextMove.append(myPlan);
engine.setTheory(nextMove);
SolveInfo info = engine.solve("selectMove(X).");
```

Alla teoria contribuisce lo stato del tavolo (belief), il proprio piano (myPlan) e la strategia (algorithm - Plan01). "selectMove(X)" unificherà il suo unico argomento con la prossima mossa da eseguire. Questa può essere nella forma:

- *"takeRouteCards"* se il piano è vuoto;
- *"takeRoute(IDR, Col, N, NUJ)"* per l'occupazione di una tratta;
- *"takeVisible(Col)"* se vuole prendere una carta scoperta;
- *"takeHidden"* se vuole prendere una carta coperta;

Infine Player controllerà la stringa corrispondente all'argomento X, eseguendone l'operazione richiesta.

Capitolo 5

Come giocare

In questo breve capitolo riassumiamo le funzioni principali della versione di Ticket To Ride da noi sviluppata.

5.0.1 Il gioco

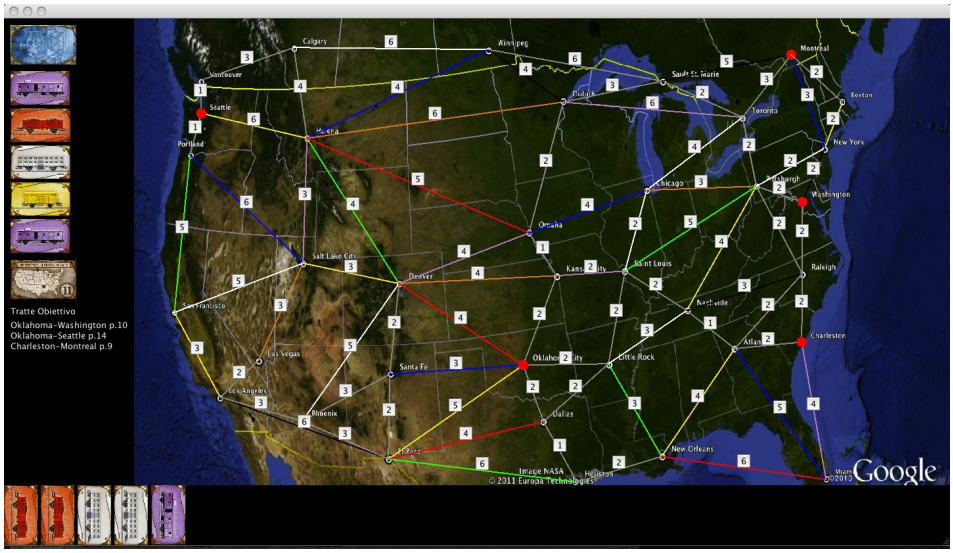

Figura 5.1: Schermata Principale

La schermata principale (fig. 4.1) è composta sostanzialmente da tre parti: la mappa, al centro, che contiene il grafo delle città e gli archi che le collegano. Ogni arco, come già visto, è colorato e possiede un pulsante tramite il quale è possibile occupare l'itinerario. Sulla colonna di sinistra (fig. 4.2) abbiamo il mazzo coperto (prima carta in alto, in blu), le 5 carte visibili che formano il mazzo scoperto, il mazzo degli obbiettivi (raffigurato tramite una carta di colore marrone/bianco) e un riepilogo delle tratte che si dovranno completare date dalle carte obbiettivo scelte. In basso invece troviamo lo spazio dove è possibile mantenere sotto osservazione le proprie carte colorate (fig. 4.3)

Figura 5.2: Mazzi e Obbiettivi

All'inizio del gioco viene richiesto di selezionare (tra le tre proposte) almeno una carta obbiettivo (fig. 4.4). Una volta che il giocatore

Figura 5.3: Le carte in possesso

umano avrà accettato la carta (o eventualmente anche più di una), le città verranno evidenziate sulla mappa attraverso un pallino rosso (fig. 4.5). A questo punto il giocatore potrà decidere se occupare una tratta, selezionando le carte in proprio possesso in misura uguale al numero di trenini necessari, oppure pescare carte dai mazzi.

Figura 5.4: Dettaglio della mappa

Al termine della propria mossa il giocatore resta in attesa del turno dei player virtuali, il quale termina con una finestra di riepilogo turni. Il gioco va avanti finché un player – sia esso l'utente o uno mantenuto da computer – non scende al di sotto della soglia dei 3 treni a disposizione. A questo punto un messaggio avvertirà l'utente di giocare l'ultima mossa in quanto questo sarà l'ultimo turno. Una volta completato verrà riproposto un riepilogo finale comprendente dei punti assegnati se gli obbiettivi sono stati raggiunti.

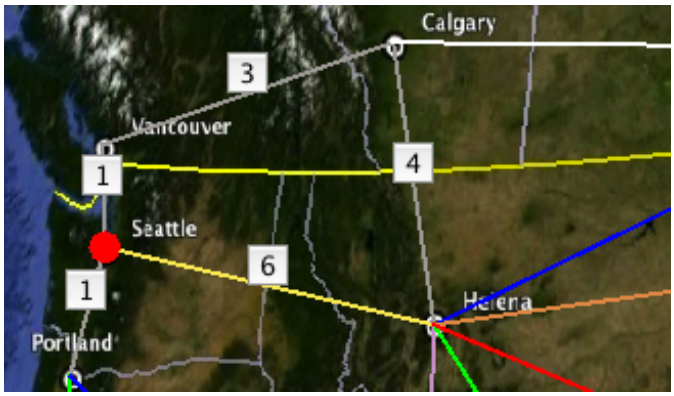

Figura 5.5: Dettaglio della mappa

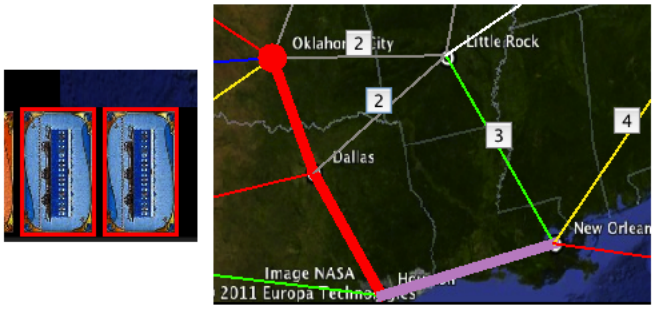

Figura 5.6: Selezione delle carta (a sinistra) e occupazione di una tratta (a destra)

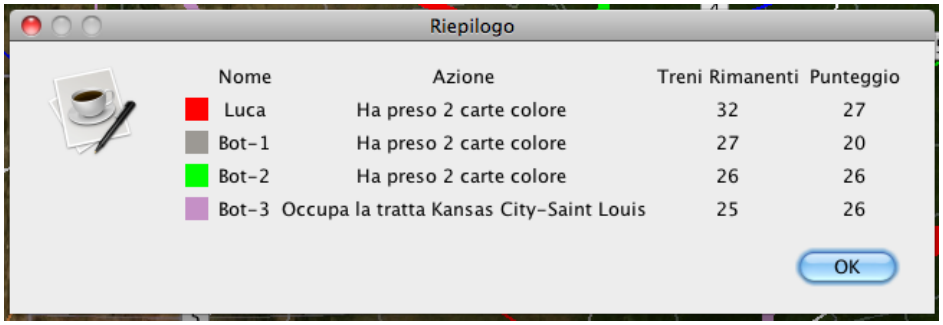

Figura 5.7: Riepilogo turno