Documentazione Caso di Studio
Ingegneria della Conoscenza
A.A.2022/2023
Agente Giocatore di Tressette

Gruppo di lavoro

- Rodolfo Pio Sassone, 719017, r.sassone3@studenti.uniba.it

< https://github.com/rodolfo-sassone/tressette >

AA 2022-2(3+1)

Introduzione

Il progetto mira a creare un agente in grado di giocare a tressette utilizzando metodi per il ragionamento e basi di conoscenza.

Il tressette è un gioco di carte, che può essere svolto con le carte napoletane o vicentine. [1]

Definizione delle fasi del gioco

Poiché potrebbero differire da quelle indicate su Wikipedia.

- Una **partita** è formata da più **round** (quanti ne servono per raggiungere la quota di punti minima che determina la squadra vincitrice);
- Ogni **round** è formato da 10 **mani** o **turni**

Regole

La variante giocata dall'agente è il tre sette muto incrociato a coppie fisse. La partita termina quando una delle due squadre colleziona almeno 31 punti.

Sommario

Per la base di conoscenza è stato utilizzato Prolog con cui è stato realizzato tutto il ragionamento. Python è stato utilizzato semplicemente per realizzare l'interfaccia utente. L'interfaccia è stata creata utilizzando pyswip che permette di aggiungere e rimuovere conoscenza e di interrogare la base di conoscenza.

Elenco argomenti di interesse

- Rappresentazione della conoscenza
- Ragionamento automatico
- Pianificazione con incertezza
- Ragionamento con incertezza

Rappresentazione della conoscenza

Conoscenza di base: carte e giocatori

Giocatori

Per specificare l'esistenza di soli quattro giocatori è stata fatta assunzione di conoscenza completa utilizzando la relazione unaria giocatore/1.

```
giocatore(me).
giocatore(compagno).
giocatore(avv1).
giocatore(avv2).
```

- "me" rappresenta l'agente
- "compagno" rappresenta il compagno di squadra dell'agente
- "avv1" e "avv2" i due avversari

Carte

Le carte sono ovviamente alla base del gioco, avevamo quindi la necessità di rappresentare per ogni carta molte caratteristiche. È stata perciò scelta la rappresentazione flessibile formata dalla terna individuo-proprietà-valore[2]

```
prop(asso_denari,type,asso).
prop(asso_denari,palo,denari).
prop(asso_denari,potere,8).
prop(asso_denari,punti,1).

prop(due_denari,type,due).
prop(due_denari,palo,denari).
prop(due_denari,potere,9).
prop(due_denari,punti,0.334).

prop(tre_denari,type,tre).
prop(tre_denari,palo,denari).
prop(tre_denari,potere,10).
prop(tre_denari,punti,0.334).

prop(dieci_denari,type,figura).
prop(dieci_denari,palo,denari).
prop(dieci_denari,potere,7).
prop(dieci_denari,punti,0.334).
```

- **type** indica il tipo della carta (asso, due, tre, figura o scartina);
- **palo** indica il palo (o seme) della carta (denari, coppe, spade o bastoni);
- **potere** indica la capacità di presa di una carta e va da 10 a 1;
- **punti** indica quanti punti vale la carta (1 per gli assi; 1/3 per due, tre e figure; 0 per le scartine).

Per quest'ultima caratteristica 1/3 è stato rappresentato come 0.334 poiché ci permette di arrotondare sempre per difetto dato che la somma di tre figure, che formano un punto, è maggiore di uno.

Ci sono poi altre caratteristiche utili al ragionamento che vedremo in seguito.

Conoscenza all'interno della partita

Tutta la conoscenza che segue viene aggiunta nel corso di ogni round e cancellata al termine dello stesso.

Possesso, palo, piombo e non_possesso

Una volta distribuite le carte l'agente viene a conoscenza delle carte che possiede (l'utente via interfaccia python dirà all'agente le carte che possiede). Il possesso è rappresentato sempre come una proprietà della carta.

```
%mazzo di prova
prop(nove_coppe,possessore,giocatore(me)).
prop(sette_coppe,possessore,giocatore(me)).
prop(tre_denari,possessore,giocatore(me)).
prop(nove_denari,possessore,giocatore(me)).
prop(tre_bastoni,possessore,giocatore(me)).
prop(otto_bastoni,possessore,giocatore(me)).
prop(quattro_bastoni,possessore,giocatore(me)).
prop(sette_bastoni,possessore,giocatore(me)).
prop(sette_spade,possessore,giocatore(me)).
prop(tre_spade,possessore,giocatore(me)).
```

Inizialmente l'agente conosce solo il possesso delle proprie carte.

Man mano che la partita va avanti l'agente può reperire nuove informazioni: un vincolo del gioco è che il giocatore di mano (il primo a tirare) stabilisce il palo del turno, gli altri giocatori dovranno rispondere obbligatoriamente a quel palo se possono, altrimenti tirano una carta di un altro palo senza alcuna speranza di presa e si dice che quel giocatore è piombo a quel palo.

Il vincolo del palo è rappresentato con la relazione palo/2, ed è aggiunta ad ogni turno alla KB in automatico dall'interfaccia python.

```
palo(0,bastoni).
palo(1,bastoni).
```

- il primo parametro indica il numero del turno (da 0 a 9);
- il secondo il palo del relativo turno.

Rappresentiamo piombo come una relazione binaria piombo/2 tra palo e giocatore.

```
piombo(P,G)

```

Quando una carta viene tirata si dice che la carta è “a terra”. Il turno durante il quale la carta è a terra viene rappresentato come un’altra caratteristica della carta stessa

```
227 prop(due_bastoni,terra,0).
228 prop(due_bastoni,possessore,giocatore(compagno)).
229
230 prop(nove_bastoni,terra,0).
231 prop(nove_bastoni,possessore,giocatore(avv1)).
232
233 prop(otto_bastoni,terra,0).
234
235 prop(sei_bastoni,terra,0).
236 prop(sei_bastoni,possessore,giocatore(avv2)).
```

È utile rappresentare anche l’informazione che un giocatore non possegga una carta. Rappresentata come una caratteristica della carta stessa così come avviene per il possesso.

```
411 %se un giocatore è piombo allora nor
412 prop(X,non_possessore,giocatore(G))
```

Accuso o Busso e mano a monte

Un giocatore che si ritrova ad avere in mano delle particolari combinazioni di carte è obbligato a fare un accuso (o a bussare) dichiarando di quale/i tra le possibili combinazioni è in possesso. Le possibili combinazioni sono la napoli, il bongioco e la corona.

Sono tutte rappresentate come relazioni binarie.

```
456 napoli(P,G) :-  prop(X,possessore,giocatore(G)), prop(X,type,asso), prop(X,palo,P),
457             prop(Y,possessore,giocatore(G)), prop(Y,type,due), prop(Y,palo,P),
458             prop(Z,possessore,giocatore(G)), prop(Z,type,tre), prop(Z,palo,P).
459
460 corona(T,G) :-
461             prop(X,possessore,giocatore(G)), prop(X,type,T), T\==scartina, T\==figura,
462             prop(Y,possessore,giocatore(G)), prop(Y,type,T), X \== Y,
463             prop(Z,possessore,giocatore(G)), prop(Z,type,T), X \== Z, Y \==Z,
464             prop(W,possessore,giocatore(G)), prop(W,type,T), X \== W, Y \== W, Z \== W, !.
465
466 corona_asso(G) :- corona(asso,G).
467 corona_due(G) :- corona(due,G).
468 corona_tre(G) :- corona(tre,G).
469
470 bongioco(T,G) :- prop(X,possessore,giocatore(G)), prop(X,type,T), T\==scartina, T\==figura,
471             prop(Y,possessore,giocatore(G)), prop(Y,type,T), X \== Y,
472             prop(Z,possessore,giocatore(G)), prop(Z,type,T), X \== Z, Y \==Z, \+ corona(T,G),!.
473
474 bongioco_asso(G) :- bongioco(asso,G).
475 bongioco_due(G) :- bongioco(due,G).
476 bongioco_tre(G) :- bongioco(tre,G).
```

Se invece il giocatore si ritrova nel proprio mazzo meno di un punto e una figura può decidere di “buttare a monte” in tal caso si rimischiano le carte.

È rappresentata da una relazione unaria con le costanti si e no. In pratica ci restituisce una risposta.

```
481 a_monte(si) :- mazzo(me,M), sumallpoint(M,S), S<1.3.
482 a_monte(no).
```

La presa

A prendere è il giocatore che tira la carta di potere più alto al palo scelto dal giocatore di mano.

La presa è rappresentata da una relazione ternaria tra il turno della presa, il giocatore che prende e il palo del turno.

```
523 presa(T,G,P) :- findall(C,prop(C,terra,T),LT), prop(D,terra,T), accPresa(LT,P,D,K), prop(K,possessore,giocatore(G)), !.
524
525 accPresa([],_,A,A).
526 accPresa([X|T],P,A,C) :- greater(P,A,X), accPresa(T,P,X,C).
527 accPresa([_|T],P,A,C) :- accPresa(T,P,A,C).
```

Carte uscite

Ultima caratteristica di una carta all'interno di un round è se è uscita o meno. Questa caratteristica differisce dalla carta a terra poiché la carta a terra si riferisce alla mano, mentre la carta uscita si riferisce al round. Concetto simile ma differente poiché durante una mano è ben differente se una carta è uscita in precedenza o è a terra nella mano corrente.

```
238 prop(due_bastoni,uscita,si).
239 prop(nove_bastoni,uscita,si).
240 prop(otto_bastoni,uscita,si).
241 prop(sei_bastoni,uscita,si).
```

Anche qui si fa assunzione di conoscenza completa: se per una carta non è presente la proprietà uscita allora non è uscita.

I pezzi o carte da tressette

Gli assi, i due e i tre vengono chiamati genericamente pezzi o carte da tressette. Rappresentati con una relazione unaria.

```
414 pezzo(C) :- prop(C,type,asso).
415 pezzo(C) :- prop(C,type,due).
416 pezzo(C) :- prop(C,type,tre).
```

Ragionamento Automatico

L'agente, dunque chiede all'utente le carte che formano il proprio mazzo, eventuali accusi di altri giocatori e le carte che, con il proseguire di ogni mano, vengono tirate. Tutte queste informazioni vengono aggiunte alla base di conoscenza e l'agente può, in questo modo, ragionare ed estrarre nuova conoscenza.

Piombo

Conoscendo il palo di ogni mano, nel momento in cui un giocatore non risponde ad una mano, l'agente può inferire che quel giocatore è piombo a quel palo.

```
419 piombo(P,G) :- palo(T,P), terra(T,LT,_), member(X,LT), prop(X,palo,Q), Q \== P, prop(X,possessore,giocatore(G)).
420
```

Di conseguenza, cosa più importante, può inferire che quel giocatore non possiede nessun'altra carta a quel palo.

```
411 %se un giocatore è piombo allora non ha nessuna carta a quel palo
412 prop(X,non_possessore,giocatore(G)) :- piombo(P,G), prop(X,palo,P), rimanenti_a(P,L,_), member(X,L).
```

Possesso per esclusione

L'informazione del "non possesso" è importante poiché ci permette di inferire per esclusione il possesso della stessa carta. Non è possibile fare ciò con NAF dato che non abbiamo la completa conoscenza del possesso delle carte e quindi non possiamo fare assunzione di mondo chiuso.

```
394 %stabilire il possesso di una carta per esclusione, sapendo che gli altri non ce l'hanno
395 prop(X,possessore,giocatore(compagno)) :- prop(X,non_possessore, giocatore(me)),
396                                          prop(X,non_possessore, giocatore(avv1)),
397                                          prop(X,non_possessore, giocatore(avv2)).
```

Carta tre

Una carta si dice "carta tre" quando, durante il round, essendo uscite tutte le carte con potere maggiore a quel palo, si ritrova ad essere la carta con il più alto potere. Ad inizio round ovviamente le "carte tre" sono solo i tre. È anche importante riconoscere di avere più carte tre in mano allo stesso palo. Questo può accadere quando le carte maggiori o sono uscite oppure sono in mano al giocatore stesso.

```
446 %carta tre, carta che prende sicuramente al suo palo
447 tre(X) :- prop(X,potere,10), !.
448 tre(X) :- min_greater(X,Y), prop(Y,uscita,si), tre(Y), !.
449 tre(X) :- min_greater(X,Y), prop(Y,possessore,giocatore(me)), tre(Y).
```

Palo della napoli

Quando un giocatore bussa napoli, non può dire qual è il palo della napoli. Bisogna quindi ragionare per esclusione rispetto alla conoscenza delle carte degli altri giocatori (se stessi compresi). Questa informazione è molto importante poiché ci rivela il possesso delle tre carte più importanti di un

determinato palo; se a bussare è stato il compagno sappiamo a quale palo abbiamo un gioco sicuro, altrimenti, cioè se a bussare è stato un avversario, sappiamo a quale palo non dovremo giocare.

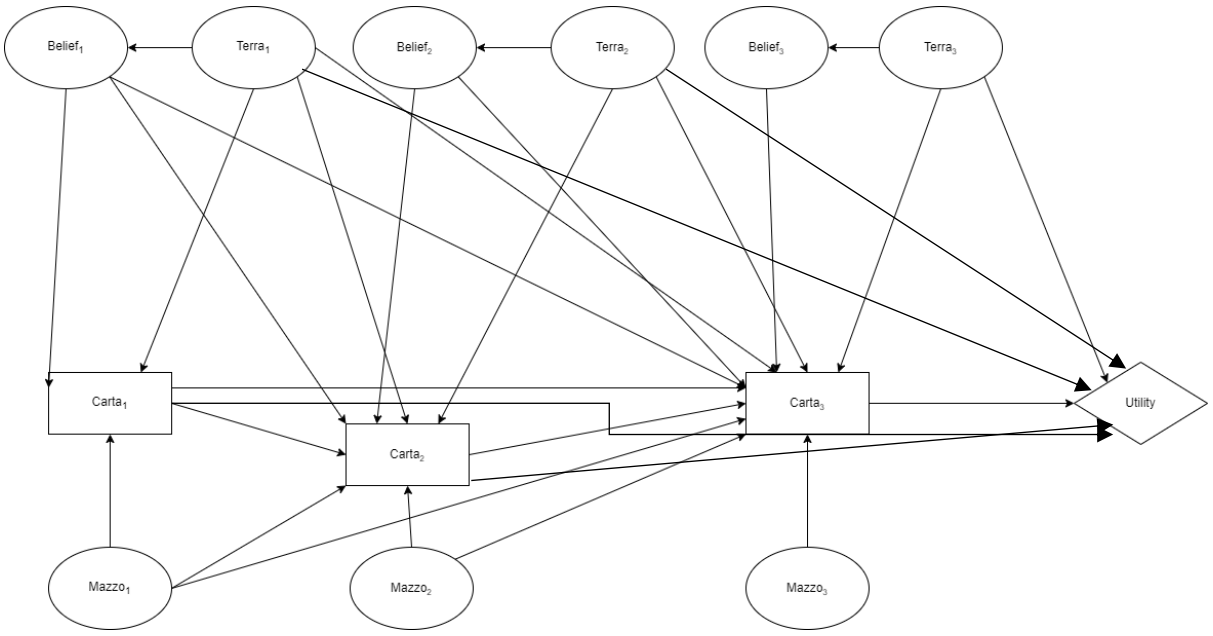
```
418 search_napoli(G,P) :- pezzo(C), prop(C,palo,Q), prop(C,possessore,giocatore(F)), F \== G, Q \== P,
419             pezzo(D), prop(D, palo, R), prop(D,possessore,giocatore(H)), H \== G, R \== Q, R \== P,
420             pezzo(E), prop(E,palo, S), prop(E,possessore,giocatore(I)), I \== G, S \== R, S \== Q, S \== P, !.
```

Pianificazione con incertezza

Chi gioca a tressette sa che la maggior parte dei punti si ottengono prendendo nelle ultime 3-4 mani, sia perché inizialmente si studia un po' "il contesto" cercando di capire le carte del compagno e degli avversari, per cui le carte di maggior valore tendono ad essere trattenute nell'attesa di poterle prendere, sia perché la "chiusura" o "rete", ovvero l'ultima presa, vale di per sé 1 punto più i punti delle carte prese. Ragion per cui è importante, per un buon giocatore di tressette, pensare e pianificare la chiusura, non può soffermarsi solo su ciò che è a terra nel turno corrente.

Idea Iniziale

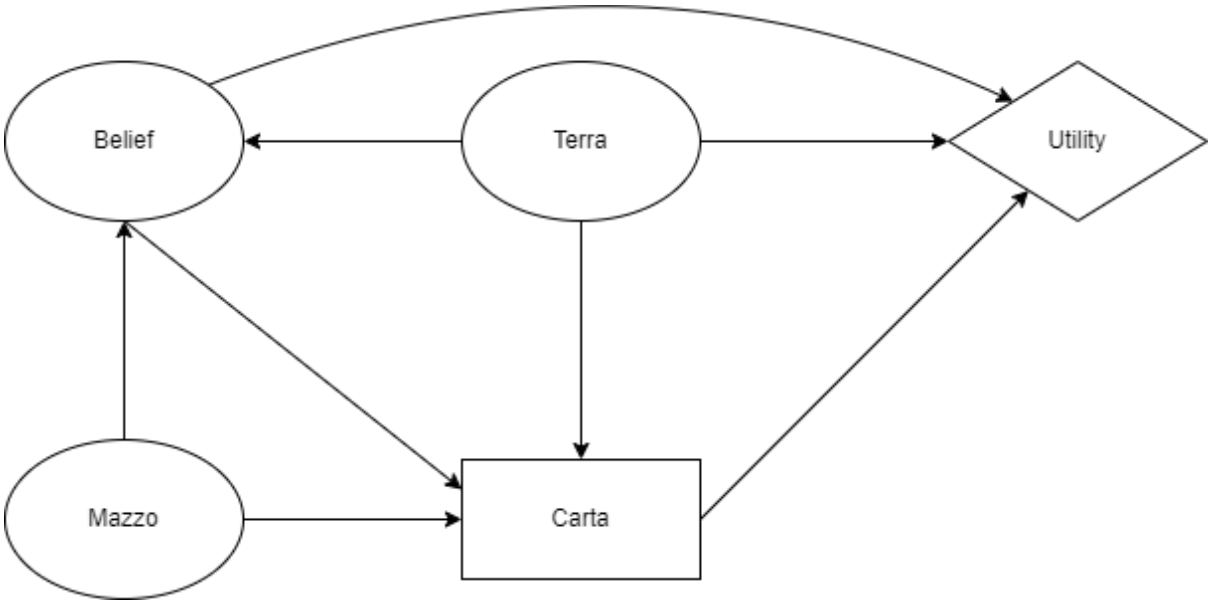
L'idea iniziale era quella di rappresentare il dominio di interesse come una no-forgetting decision network[3] con dieci nodi decisionali, uno per ogni carta da tirare, e le variabili $terra_i$ (a rappresentare le carte a terra nel turno i), $mazzo_i$ (a rappresentare le carte a disposizione al turno i) e $belief_i$ (a rappresentare le carte rimanenti, le carte in possesso degli altri giocatori, e quant'altro fosse utile al ragionamento) come chance node. Tuttavia, data la scarsezza di conoscenza iniziale risulta molto complesso calcolare quale sia al turno 1, ad esempio, il possibile valore di $terra_{10}$ poiché non sappiamo chi possiede quali carte. In pratica risulta difficile calcolare quali sono le *probabili* mosse degli avversari (e/o del compagno) se non sappiamo neanche quali sono le loro *possibili* mosse.



a Esempio con sole tre carte

Schema finale

Semplificando abbiamo ottenuto una single-stage decision network[3]. Ciononostante, i nodi belief e mazzo, conservando informazioni come le carte rimanenti, i relativi punti, il possesso o il non possesso di una carta e le informazioni rispetto ad ogni carta nel mazzo, fanno sì che la valutazione della carta da tirare non si fermi al singolo turno ma guardi anche al futuro. Praticamente tutte le informazioni dei diversi nodi $belief_i$ sono collassate in belief che si aggiorna ad ogni turno con nuova conoscenza proveniente dalle carte a terra e da quelle disponibili. Per quanto riguarda terra, possiamo provare, come vedremo, a predire se una carta cade nel turno attuale, o, al massimo, nel prossimo.



- **terra** contiene tutte le carte a terra nel turno;
- **mazzo** contiene tutte le carte disponibili all'agente;
- **belief** contiene tutte le info utili al ragionamento es:
    - carte tre,
    - stima dei punti,
    - punti rimanenti,
    - possesso,
    - carte uscite/rimanenti,
    - ecc.

Decisioni di Progetto

Ad ogni turno, quindi, viene calcolata l'utility di ogni carta a disposizione dell'agente.

Per calcolare l'utility è stata ideata una formula che prende spunto dal calcolo del Q-value di una policy per agenti di planning ad orizzonte indefinito o infinito[4].

Possiamo scomporre la formula in due parti:

Utility nel turno attuale

$$\mathrm{R} = \max\Big(P\big(presa(c)\big), P(presa\ compagno)\Big) * \big(punti\ a\ terra + stima\ punti(c)\big)$$

P(presa(c)) è la probabilità di prendere con la carta c; *P(presa compagno)* è la probabilità di prendere del compagno (se conosciuta altrimenti è 0); *punti a terra* è la somma delle carte già a terra; *stima punti* è la stima dei punti che cadranno all'interno dello stesso turno, carta giocata dall'agente inclusa (es. l'agente deve giocare per primo, e quindi devono cadere altre tre carte).

```
prob_presa(C,P,_,1) :- prop(C,palo,P), tre(C), !.
prob_presa(C,P,T,1) :-  prop(C,palo,P), findall(K, prop(K,terra,T), LT), all_greater(C,P,LT,L), conta(LT,NT), NT == 3, conta(L,N), N == 0, !.   %Tutte le carte a terra sono minori di C
prob_presa(C,P,T,0) :-  prop(C,palo,P), findall(K, prop(K,terra,T), LT), all_greater(C,P,LT,L), conta(L,N), N>0, !. %C'è già a terra una carta più alta di C
prob_presa(C,P,_,0) :- prop(C,palo,Q), Q \== P, !.
prob_presa(C,P,T,Prob) :- prop(C,palo,P), rimanenti_a(P,_,NR), terra(T,LT,_), per_palo(P,LT,LTP), conta(LTP,NTP), rimanenti_superiori(C,_,NRS), disponibili(P,_,ND),
                          NRIG is NR-NRS-ND-NTP, Prob is NRIG/(NR-ND-NTP), !.
```

La probabilità di presa di una carta *c* è calcolata dividendo il numero di carte superiori a *c* non ancora uscite al palo della mano per il numero di carte non ancora uscite al palo della mano.

La probabilità di presa del compagno è calcolata allo stesso modo purché conosciamo la carta già tirata dal compagno o una carta a sua disposizione.

```
prob_presaComp(T,P,Prob) :- carta_compagno(T,C), prob_presa(C,P,T,Prob), !.
prob_presaComp(_,P,0) :- disp_comp(P,_,N), N == 0, !.
prob_presaComp(T,P,Prob) :- disp_comp(P,L,_), member(X,L), max_potere(L,X,C), prob_presa(C,P,T,Prob).
```

punti a terra è calcolata semplicemente sommando tutti i punti a terra nel momento in cui l'agente tira la carta.

La *stima dei punti* che potrebbero uscire è calcolata in generale come segue:

$$stima\ punti(c) = punti(c) + \big(\max\big(punti(carte\ rimanenti)\big) * P(cade)\big) + (0.2 * n)$$

punti(c) sono i punti della carta che stiamo valutando di tirare e *punti(carte rimanenti)* è una lista con i punti di tutte le carte rimanenti al palo del turno ma non disponibili all'agente.

P(cade) calcola la probabilità che una carta ad un determinato palo cada, ovvero che un avversario sia costretto a tirarla (il nostro compagno, se può, non esiterà a darcela). Dato che l'avversario è restio a tirare carte di valore, possiamo assumere che *P(cade)* sia la probabilità che cada la carta con il più alto valore e quindi, più in generale, l'asso.

La media dei punti delle carte nel tressette è 0.27, in questo caso arrotondato per difetto a 0.2; *n* è il numero di giocatori che devono tirare dopo l'agente. Tuttavia, per evitare una sovrastima importante, quando la probabilità che la carta cada è "buona" (varia a seconda di quando l'agente tira, in generale possiamo dire maggiore o uguale al 65%) *n* viene posto uguale a 1.

Utility futuro

Dividiamo questa parte in altre due parti:

- Delta tre
  Il delta tre è la variazione di carte tre all'interno del mazzo dell'agente. Come detto precedentemente, le prese più importanti sono le ultime 3-4; è quindi fondamentale preoccuparsi di avere le carte a disposizione per poter prendere in quelle ultime mani.

  $$DT = \big(numero\ future\ tre - numero\ tre\ disponbili - tre(c)\big) * coeff$$

  Le *tre disponibili* sono tutte le carte tre in mano all'agente.
  Le *future tre* sono tutte le carte tre in mano all'agente più le carte che diventeranno tre al prossimo turno poiché al turno attuale sono cadute tutte le carte rimanenti maggiori ad una che l'agente ha in mano.
  La funzione *tre(c)* vale 1 se *c* è carta tre e 0 altrimenti.
  Il tutto è infine moltiplicato per un coefficiente che va a smorzare il peso del delta tre all'interno di tutta l'utility per fare in modo che non abbia un peso eccessivo. Questo coefficiente varia in base ad alcune condizioni:
  - se l'asso al palo di *c* (che è tre) è uscito *coeff* è 0.2: se l'asso è uscito con buone probabilità le carte a quel palo sono terminate, giocare la carta tre a quel palo non danneggia di molto la probabilità di prendere in futuro poiché a quel palo non giocherà più nessuno;
  - se l'asso al palo di *c* (che è tre) non è uscito *coeff* è 0.65: se l'asso non è uscito, le probabilità che ci sia ancora gioco a quel palo sono alte, non dobbiamo sprecare *c*, d'altra parte non possiamo neanche tenerla in mano e non giocarla mai;
  - se *c* non è tre *coeff* è 0.8: se c non è tre la differenza è positiva o nulla (*future tre* >= *tre disponibili*).

- Punti futuri

  $$V = \big(P(presa\ rimanenti) - P(presa\ rimanenti(c))\big) * punti\ rimanenti$$

  P(presa rimanenti) è la probabilità di prendere le carte rimanenti ed è la media della probabilità di presa dei quattro pali. Per ogni palo la probabilità di presa, avendo una carta tre a quel palo, è la probabilità che si giochi a quel palo; 0 se non abbiamo una carta tre a quel palo.
  P(presa rimanenti(c)) è la probabilità di presa in futuro della carta *c*. È calcolata controllando se *c* è futura tre e, in caso affermativo, la probabilità di presa è calcolata come la probabilità che si giochi a quel palo (come sopra) e poi divisa per quattro per "normalizzare" il risultato al risultato di P(presa rimanenti); in caso non sia *futura tre* è 0.
  punti rimanenti è la somma dei punti delle carte non ancora uscite meno i punti della carta che stiamo valutando di tirare.

Utility finale

Come detto sopra, è stata presa ispirazione dalla formula del Q-value; questo perché c'è la necessità di dare peso al futuro e ragionare anche sulle prossime mosse nonostante l'impossibilità di conoscere le mosse di avversari e compagno, in modo simile un agente di planning ad orizzonte indefinito o infinito deve preoccuparsi, una volta entrato in uno stato, di quali sono le sue future possibilità partendo da quello stato. È stata presa ispirazione anche per la nomenclatura: l'utility attuale è chiamata R come i reward e l'utility futura è chiamata V come l'expected value di una policy.

$$U = R + DT + V$$

Dove *R*, *DT* e *V* sono definite come sopra.

Infine l'agente ad ogni turno seleziona la carta a disposizione con il maggior valore di utility.

```
553 seleziona(T,C) :- disponibili(M,_), accSeleziona(T,-10,M,_,C).
554
555 accSeleziona(_,_,[],A,A).
556 accSeleziona(T,U,[X|R],_,C) :- tira(T,X,Unew), Unew > U, accSeleziona(T,Unew,R,X,C), !.
557 accSeleziona(T,U,[_|R],A,C) :- accSeleziona(T,U,R,A,C).
```

Valutazione

Per la valutazione sono stati eseguiti dieci round simulati.

Le simulazioni sono state svolte chiedendo a ChatGPT di distribuire le carte, così da avere quattro mazzi, e poi un solo utente ha giocato per tutti e tre gli altri giocatori.

Dalle simulazioni[7] emerge che nei casi in cui l'agente ha un ottimo mazzo si comporta molto bene. Ciò significa che, quando la partita è di fatto nelle mani dell'agente, quando dalle sue azioni dipende l'andamento della partita, prende buone scelte: nelle simulazioni in cui l'agente possedeva il mazzo M3 (round 1 e 6) con un bongioco di tre riesce, nel caso peggiore (il compagno non ha nulla: mazzo M2) a conquistare 4 punti praticamente da solo; e nel caso migliore (compagno con mazzo M1) conquista, insieme al compagno ben 10 punti. Quando il mazzo è nella media riesce a giocare abbastanza bene e quando il mazzo contiene poche carte di valore, quindi c'è poco da pianificare, scarta carte di poco valore. In totale l'agente ha collezionato 63 punti su 110 vincendo 7 round su 10.

Ragionamento con incertezza

Per utilizzare un metodo trattato nel corso e avere un metro di paragone per valutare la Decision Network è stato creato un nuovo ragionamento utilizzando una Belief Network[5].

Una Belief network è un grafo che rappresenta le dipendenze tra variabili casuali. Quest'ultime sono rappresentate tramite nodi e gli archi orientati rappresentano le dipendenze.

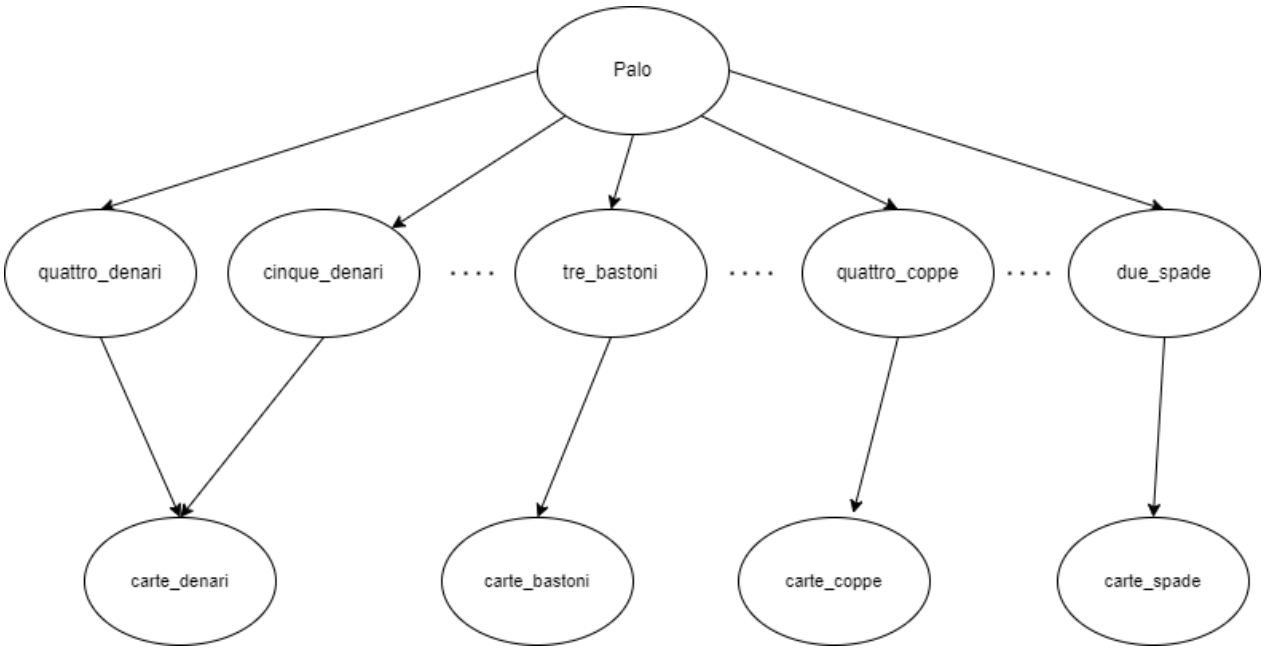
La policy è greedy: se possiamo prendere prendiamo con la carta più bassa che abbiamo, altrimenti scartiamo una delle carte di minor valore.

Idea

L'idea è di calcolare le carte che più probabilmente usciranno nel corso della mano corrente, ovvero predire le prossime carte, così da poter tirare una carta che probabilmente prenderà. Per fare ciò è stata realizzata una Belief Network avente le seguenti variabili:

- Palo: con dominio [0, 1, 2 , 3], rispettivamente denari, coppe, spade, bastoni. A rappresentare la probabilità che si giochi ad un determinato palo;
- Una variabile booleana per ognuna delle 40 carte del mazzo. Rappresenta la probabilità che la carta esca;
- Una variabile per ogni palo con dominio [0,1,2,3,4,5,6,7,8,9], rispettivamente quattro, cinque, sei, sette, otto, nove, dieci, asso, due, tre (la relazione tra il numero e la carta è potere della carta – 1). Questa variabile serve a calcolare la probabilità che una carta esca sapendo che altre carte dello stesso palo sono uscite.

$$P(carte\_denari | quattro_{denari} = False, otto_{denari} = False)$$



Esempio con alcune carte

Vista la politica attuata dall'agente, infatti, non ha senso calcolare le probabilità che esca una qualsiasi carta, ma solamente delle carte del palo corrente così che possa predire la sua probabilità di presa.

Per realizzare la BN è stato usato il codice di AIPython[6]messo a disposizione dal libro.

La probabilità a priori per ogni valore di *palo* è 1/4.

Per le carte le probabilità sono state calcolate come segue:

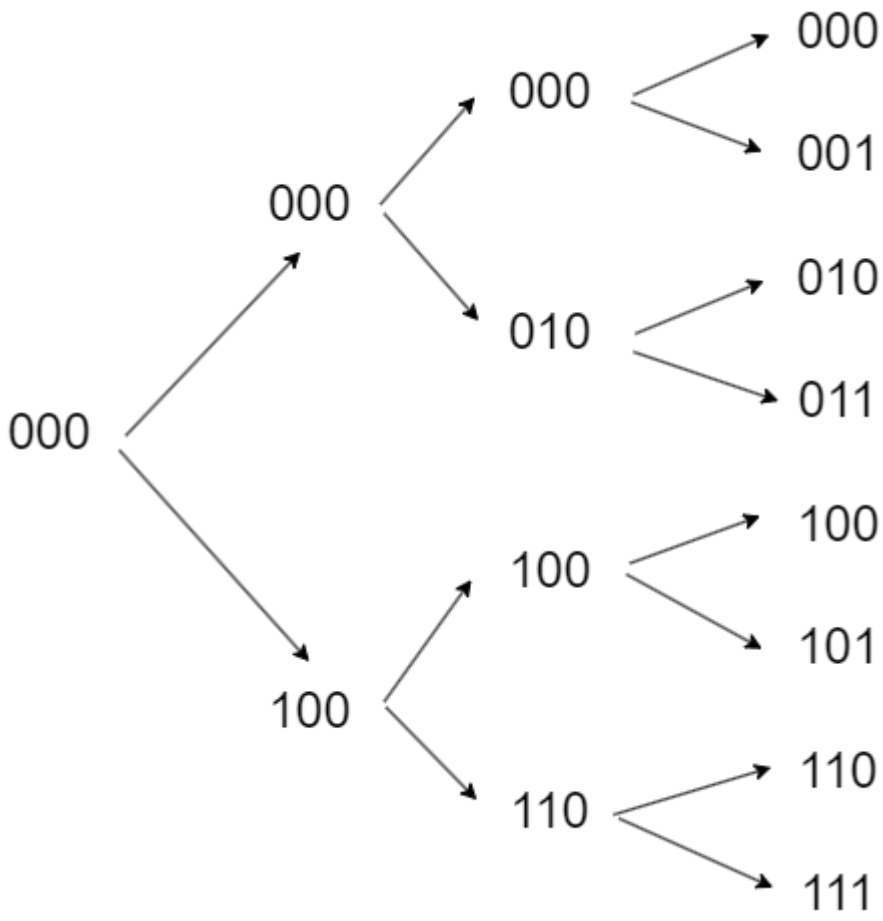
Ad ogni numero di carta è stata assegnata una probabilità generale di uscire; quest'ultima è stata moltiplicata per un coefficiente diverso a seconda se il valore di palo corrisponde a quello della carta o meno.

```
quattro = 0.25
cinque = 0.15
sei = 0.12
sette = 0.1
otto = 0.09
nove = 0.09
dieci = 0.09
asso = 0.02
due = 0.07
tre = 0.02

coeff_palo = 0.85
coeff_notPalo = 0.05
def carta_palo(carta):
    return carta*coeff_palo

def carta_notPalo(carta):
    return carta*coeff_notPalo

def get_prob(var,pars,palo,value):
      if palo == Palo.DENARI or palo == 0:
            f = Prob(var,pars,[[(1-carta_palo(value)), carta_palo(value)],
                               [(1-carta_notPalo(value)), carta_notPalo(value)],
                               [(1-carta_notPalo(value)), carta_notPalo(value)],
                               [(1-carta_notPalo(value)), carta_notPalo(value)]])

      elif palo == Palo.COPPE or palo == 1:
           f = Prob(var,pars,[[(1-carta_notPalo(value)), carta_notPalo(value)],
                               [(1-carta_palo(value)), carta_palo(value)],
                               [(1-carta_notPalo(value)), carta_notPalo(value)],
                               [(1-carta_notPalo(value)), carta_notPalo(value)]])

      elif palo == Palo.SPADE or palo == 2:
           f = Prob(var,pars,[[(1-carta_notPalo(value)), carta_notPalo(value)],
                               [(1-carta_notPalo(value)), carta_notPalo(value)],
                               [(1-carta_palo(value)), carta_palo(value)],
                               [(1-carta_notPalo(value)), carta_notPalo(value)]])

      elif palo == Palo.BASTONI or palo == 3:
            f = Prob(var,pars,[[(1-carta_notPalo(value)), carta_notPalo(value)],
                               [(1-carta_notPalo(value)), carta_notPalo(value)],
                               [(1-carta_notPalo(value)), carta_notPalo(value)],
                               [(1-carta_palo(value)), carta_palo(value)]])

      return f
```

La probabilità di *carta_denari*, *carta_coppe*, *carta_spade* e *carta_bastoni* sostanzialmente è la probabilità che una carta di quel palo cada ed è per ogni carta che può cadere 1/n dove n è il numero di carte che possono cadere e 0 per le altre. L'obbiettivo è di conoscere la carta che più probabilmente cadrà, ovvero predire la prossima carta.

Per creare la Tabular CPD di queste variabili, fondamentale per creare un oggetto di tipo Prob, è stata utilizzato un array di booleani a rappresentare tutte le possibili combinazioni di verità delle variabili padre e una funzione ricorsiva *get_row()* che crea un'assegnazione se lo spazio del dominio è uno e divide in due il dominio e gli assegna un sottoinsieme delle possibili combinazioni altrimenti (metà dell'insieme che le viene passato).

Per dividere in due l'insieme delle possibili combinazioni si fa una copia dell'array e partendo da sinistra si pone a *True* un valore così da avere due array; per tenere traccia dell'ultima cifra che è stata cambiata si utilizza un indice. Fatto ciò, si richiama due volte get_row() passando ad uno l'array come è stato ricevuto e all'altro la copia modificata.



Un esempio con numero binario a tre cifre

Per quanto riguarda *size*, ovvero la dimensione del dominio, essendo una potenza di due (poiché tutte le variabili padre sono booleane) è sicuramente pari e quindi nessun problema con la divisione per due.

Di seguito il codice sopra descritto.

```
157   def get_row(array, i, size):
158       if size == 1:
159           value = []
160           for b in reversed(array):
161               if b:
162                   c = np.count_nonzero(array == True)
163                   value.append(float(1/c))
164               else:
165                   value.append(float(0))
166
167           return value
168
169       elif size%2 == 0:
170           i = i+1
171
172           array2 = np.copy(array)
173           array2[i] = True
174
175           row1 = get_row(array, i, size/2)
176           row2 = get_row(array2, i, size/2)
177
178       return [row1,row2]
179
180
181   def get_TabularCPD(var):
182       array1 = np.zeros(var.get_size(), dtype=bool)
183       array2 = np.zeros(var.get_size(), dtype=bool)
184       i = 0
185       array2[i] = True
186
187       row1 = get_row(array1, i, (2**var.get_size())/2)
188       row2 = get_row(array2, i, (2**var.get_size())/2)
189
190       return [row1,row2]
```

In questo modo si ottiene esattamente la rappresentazione della Tabella così come la richiede l'oggetto Prob.

Creata la BN è stata creata la classe TressetteVE che discende dalla classe VE di AIPython. VE sta per Variable Elimination che è il metodo utilizzato dalla classe per fare inferenza esatta sulla BN.

TressetteVE oltre i metodi ereditati dalla classe padre contiene parametri e metodi specifici per il ragionamento nel tressette che sono stati inseriti all'interno della stessa classe per comodità. In particolare, il metodo query è stato modificato affinché restituisca direttamente la carta da tirare. Calcola le carte che più probabilmente usciranno, calcola quindi il potere a terra atteso, e se c'è possibilità di presa restituisce la carta con il potere minore che prende, altrimenti restituisce una carta con il più basso valore di punti possibile.

Valutazione

La valutazione si è tenuta allo stesso modo della DN. Nello specifico sono stati effettuati dieci round simulati.

Le simulazioni sono state effettuate con gli stessi mazzi utilizzati nelle simulazioni della DN.

Dalle simulazioni[7] emerge che l'agente con un buon mazzo non è in grado di sfruttare a pieno il potenziale di quest'ultimo, sprecando carte di valore e permettendo agli avversari di giocare liberamente, a maggior ragione quando il compagno, possedendo un pessimo mazzo non può essere di alcun aiuto. Nel caso medio, quando il compagno ha un buon mazzo risulta, guardando i punti, efficace non sprecando carte di valore e prendendo quando può. Nel caso peggiore, in cui non c'è nulla da pianificare, scarta la carta di minor valore, ciò gli consente di conservare quel poco che ha. In totale l'agente ha collezionato 61 punti su 110 vincendo 7 round su 10.

Conclusioni

Confronto

La differenza tra i due metodi si accentua quando c'è da pianificare, quando pensare alle mosse successive può fare la differenza, ancor di più quando il compagno non riesce ad essere d'aiuto causa carte di poco valore; si assottiglia, invece e di parecchio, quando non avendo carte di valore in mano pensare al futuro è inutile.

Sviluppi futuri

Il ragionamento basato sulla decision network potrebbe essere ulteriormente migliorato aggiungendo al ragionamento l'ordine dei turni effettivi di ogni giocatore così da poter sfruttare quest'altra meccanica (es. Sono il secondo a tirare, in mano ho asso e due del palo del turno corrente e so che il tre è in mano al giocatore che ha tirato per primo, quindi posso giocarmi l'asso). Così facendo ha anche molto più senso la conoscenza del possesso degli avversari che ad oggi è usata praticamente solo per derivare il non possesso del compagno.

Il ragionamento che sfrutta la belief network invece è molto migliorabile: non gioca con il compagno, si potrebbe introdurre la meccanica dei turni come nella DN e si potrebbe aggiungere del ragionamento riguardo i punti delle carte cercando di massimizzare i punti della presa.

In entrambi i casi si potrebbe aggiungere la possibilità di comunicare con il compagno così da poter giocare a modalità del tressette diverse da quella muta.

Un altro possibile sviluppo per il ragionamento con decision network sarebbe quello di utilizzare la belief network ("spogliata" di tutta la parte di ragionamento) per predire le prossime carte.

L'interfaccia utente sarebbe assolutamente da migliorare: inserire le carte risulta stressante e per niente appagante.

Riferimenti Bibliografici

[1] https://it.wikipedia.org/wiki/Tressette

[2] https://artint.info/3e/html/ArtInt3e.Ch16.S1.html

[3] https://artint.info/3e/html/ArtInt3e.Ch12.S2.html

[4] https://artint.info/3e/html/ArtInt3e.Ch12.S5.html

[5] https://artint.info/3e/html/ArtInt3e.Ch9.S3.html

[6] https://artint.info/AIPython/

[7] Tabella valutazione