POLITECNICO DI MILANO
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

PATROLLING SECURITY GAMES: ALLARMI SPAZIALMENTE E FUNZIONALMENTE IMPERFETTI

Relatore: Prof. Nicola Gatti
Correlatore: Prof. Nicola Basilico
Correlatore: Ing. Giuseppe De Nittis

Tesi di Laurea di:
Alessandro Colombo, matricola 816509

Anno Accademico 2014-2015

Ci sarebbero tante persone a cui dedicare questa tesi, benchè una tesi non sia proprio il massimo come dedica. La dedico a tutti quelli che ogni giorno si sforzano per capire le cose, per ragionarci, per sbatterci la testa, bestemmiarci su, sconfortarsi, riprendersi, bestemmiare di nuovo, e alla fine, dopo tante sonore bestemmie, arrivare ai concetti che cercano di insegnarci. Per poi ripetere il tutto di fronte a un nuovo argomento o a una nuova sfida.

La dedico a tutti coloro che ancora sperano e ancora sognano di fare qualcosa di importante. A coloro che scrivono, che leggono, che parlano, e riescono a discutere di qualcosa di più intelligente di chi sia stato nominato al Grande Fratello.

La dedico a tutti coloro che lottano per ottenere quello che vogliono, ben sapendo che spesso non è una lotta semplice. Perchè, anche se è banale, è proprio vero che le cose più importanti non si ottengono facilmente. Vedi una laurea in ingegneria, ad esempio.

La dedico a tutti coloro che ridono, scherzano e non prendono tutto sul serio. A quelli che al terzo bicchiere vedono la madonna, e al quinto iniziano a parlarci, e al settimo anche a provarci. Svegliandosi la mattina con un mal di testa che è l'ira di Dio.

La dedico a tutte le persone straordinarie che esistono, e che portano la luce in questo triste e grigio mondo. Quelle persone che riescono a far spuntare il sole in un giorno di pioggia, e a far vedere le stelle anche in una notte buia e senza Luna.

Tutto questo, anche se poco, è per voi.

Sommario

Il soldato Ryan è di pattuglia in una base militare, posizionata in territorio nemico. L'intelligence ha scoperto che una squadra di guastatori è stata incaricata di infiltrarsi nella base e di distruggere dei possibili edifici chiave, quali il deposito di munizioni, il quartier generale, e la sala radio. Quale percorso dovrebbe scegliere il soldato, per proteggere al meglio gli obiettivi sensibili?

Questo problema rientra sotto molti campi. Intelligenza artificiale, sistemi multiagente ma, soprattutto, teoria dei giochi. In particolare, si può modellare il problema come un Patrolling Security Game, *ovvero un particolare gioco nel quale il difensore e l'attaccante devono, rispettivamente, difendere o attaccare determinati obiettivi. Ogni obiettivo è rappresentato come un nodo su un grafo, con un valore e un tempo necessario per l'attacco, ed è collegato agli altri obiettivi tramite degli archi pesati in base alla distanza o al tempo necessario per raggiungerli.*

Lo scopo della tesi è estendere suddetta categoria di giochi introducendo dei sensori di rilevamento, che mandano un segnale di allarme al difensore in caso di attacco. Per rendere più realistico il modello, i sensori sono affetti da un'incertezza spaziale (ovvero coprono una grossa area, e dunque l'allarme non è riconducibile a un singolo target, bensì ad un insieme di target) e un'incertezza funzionale (è presente una percentuale di falso negativo, ovvero l'allarme potrebbe non scattare durante un attacco nel α *% dei casi).*

Come verrà poi mostrato, il problema di trovare la migliore strategia di pattugliamento è NP-HARD, e verrà dunque analizzata la scalabilità dell'algoritmo di risoluzione esatta. Verranno inoltre proposte diverse euristiche per risolvere in tempo polinomiale il problema, verificandone la perdita di valore rispetto all'algoritmo esatto e il risparmio di effort computazionale.

Abstract

Private Ryan is on patrol in a military base, located in enemy territory. He has been told that an enemy squad is about to attack the camp, with the objective to destroy some important resources, like the ammunition depot, the headquarters or the communication tower. How should the soldier choose his patrolling path to best defend these sensitive targets?

This problem is related to many field of interest: artificial intelligence, multiagent system and, overall, game theory. In particular, this problem can be modeled with a Patrolling Security Game*, where a defender and an attacker must, respectively, defend or attack some targets. Every target is a node on a graph, and has a value and a deadline, i.e. the time needed to be destroyed, and is linked to the other targets by some arcs, whose weights depend on the distance or the time needed to be reached.*

In our thesis we want to extend this model introducing some sensors that will send an alarm signal to the defender if a target is under attack. To make them more realistic, the sensors are affected by spatial and functional uncertainty. To be more precise, they cover more targets at once, so a single signal cannot be reconducted to a single target, but to the whole set, and they could not send an alarm signal, even if one of the targets is under attack, in the α *% of the cases.*

We will then show that the problem to find an optimal patrolling route is NP-HARD, so we will analyze the scalability of the exact algorithm, and we will propose some heuristics to solve the problem in polynomial time, analyzing their losses with respect to the exact one.

Ringraziamenti

Mi servirebbe un'intera tesi per ringraziare tutte le persone che dovrei ringraziare. Purtroppo lo spazio a disposizione è poco, e cercherò di condensarvi tutti qui dentro. Se vi ho dimenticati per iscritto, non preoccupatevi che non vi ho dimenticati nella mia testa e nel mio cuore.

Partiamo dai parenti. Grazie mamma, papà, Dario, www e carotino, anche se non c'è più, per avermi sopportato e avermi assecondato in tutte le mie manie, particolarità e, probabilmente, pazzie. Grazie zia Tina, zio Paolo, zia Lella, Marcello, Goffry, zia Hilde, zio Tonno, Fra e Cri per essermi sempre stati accanto, nonostante il mio caratteraccio. E grazie nonna Rosi per tutto quello che hai sempre fatto per me. Spero tanto che esista un aldilà e tu possa vedermi diventare un ingegnere, come hai sempre voluto, anche da lassù.

Ringrazio i miei amici di una vita, con cui ho condiviso gioie, dolori, risate, bestemmie e maledizioni a varie società aliene. Grazie a Genti, per esserci sempre stato, e per aver sempre proposto le migliori cazzate che abbiamo mai fatto. Grazie a Saia, per le sue storie, i suoi aneddoti, le sue frasi, e per averci ficcato tutti insieme nei peggiori casini, che sono sempre le avventure migliori e più divertenti. Grazie a Giga, per avermi insegnato a non mollare mai, ad impegnarsi sempre, e a mangiare 10 pacchetti di patatine e di briochine in una sola serata. E grazie anche a quel coglione di Vemmaman, che prima o poi ritornerà sulla strada della ragione.

Ringrazio tutta la gente meravigliosa con cui ho condiviso l'università. Grazie a Carmi per avermi insegnato la "sciallezza". Grazie a GC per tutte le stupidate che abbiamo combinato insieme, e per tutti i film trash che siamo riusciti a spararci. Grazie a Libero, il grande filosofo cultore di "figa e taleggio". Grazie a zio Bob, che mi ha spiegato che l'unico comunismo reale è quello in cui ciò che è tuo è mio e ciò che è mio è mio. Grazie a Chetta,

per la sua simpatia. Grazie a Balzi, che è praticamente il mio fratello malvagio. Grazie a Gio, per tutti i nostri discorsi filosofici, sulla vita, l'universo e nanananana... batman! Grazie a Dario, che non ammetterà mai la sua omosessualità latente. Grazie a Lory e a Tamy, per le nostre sfide a giochini sul cellulare in cui non ho mai vinto nemmeno una volta. Grazie a Mattia per tutto l'aiuto che mi ha sempre dato, e per tutte le bestemmie e gli insulti che mi ha insegnato. Grazie a Max, Nick e Gabri, con cui ho condiviso di tutto e di più. In modo etero, precisiamo. Quasi sempre etero, per essere sinceri. Ok, ammettiamolo, in realtà conviviamo felicemente insieme, e stiamo pensando di adottare due o tre figli. Grazie a Selenia, per il tempo che abbiamo passato insieme. Grazie a Davide, l'uomo più intelligente che conosca, e i cui capelli sono fan sfegatati di Dragonball. Grazie a Elisabetta, per la sua carica illimitata di energia, che ha permesso a lei, e ha costretto me, a lavorare fino alle 5 di notte. Grazie a Daniela, di cui Davide sta ancora aspettando la famosa ruota delle armi, che alla fine non è mai stata nemmeno usata...

Grazie anche a quei professori, severi, simpatici, forse pazzi ma sempre giusti, che mi hanno insegnato tutto ciò di cui avevo bisogno. Mentre agli altri auguro una grande carriera nell'ambito agrario, con specializzazione nell'agricoltura manuale tradizionale.

Grazie a NicolaB, NicolaG e Giuseppe per la loro pazienza e il loro supporto nello scrivere questa tesi. Penso che i vostri filtri antispam saranno contenti di non ricevere più dalle 3 alle 10 mail al giorno chiedendo consigli e spiegazioni su ogni singolo dettaglio.

Arriviamo ora ai ringraziamenti strani. Grazie a Google, Wikipedia e Stackoverflow che hanno ridotto al tempo di qualche click la risoluzione di problemi che mi avrebbero portato via giornate intere. Grazie a Youtube e AdblockPlus, per avermi consentito di sentire gratis qualsiasi canzone volessi, senza un minimo di pubblicità. Grazie a Tex, Dylan, Nathan, Martin, Paperone, Paperino, Batman, Deadpool, Ratchet, Jak, Kratos, Drake, Goku, Vegeta, Naruto e tutti quei personaggi che vivono nel mondo della nostra immaginazione, per avermi comunque fatto passare tante ore in loro compagnia, e per avermi, comunque, insegnato tanto.

E, infine, poichè dulcis in fundo, *grazie a quella stupenda creatura che è riuscita a farmi vedere di nuovo, a sentire di nuovo, e ad amare di nuovo. La persona che mi ha ricordato cosa voglia dire effettivamente vivere, e non solo esistere. La persona che regge il senso di tutto questo mondo, e forse dell'intero universo.*

Grazie Sarah. Ti amo.

Indice

Sommario **I**

Abstract **III**

Ringraziamenti **V**

1 Introduzione **3**
1.1 Area di ricerca 3
1.2 Contributi 4
1.3 Struttura della tesi 4

2 Stato dell'arte **7**
2.1 Teoria dei giochi 7
2.1.1 Cos'è un gioco? 7
2.1.2 Giochi in forma estesa 9
2.1.3 Giochi in forma strategica 13
2.1.4 Risoluzione di giochi in forma strategica 13
2.2 Complessità Algoritmica 18
2.2.1 Qualche definizione 18
2.2.2 Esponenziale vs Polinomiale 19
2.2.3 $\mathcal{P}$ vs $\mathcal{NP}$ 20
2.2.4 Il problema del millennio 22
2.3 Patrolling Security Games 22
2.3.1 PSG: riduzione a strategie deterministiche 24
2.3.2 PSG: introduzione degli allarmi 24

3 Formulazione del problema **27**
3.1 Formulazione del modello 27
3.1.1 PSG: allarmi con falsi negativi 27
3.1.2 PSG in forma estesa 28
3.1.3 PG: best placement vs patrolling route 31

3.1.4 SRG-v: risoluzione in forma strategica 31
3.1.5 SRG-v: complessità 33
3.2 Obiettivi della ricerca 33

4 Algoritmi **35**
4.1 SRG: Ricerca dei Covering Set 35
4.1.1 Algoritmo esatto: dynamic programming 36
4.1.2 Euristica : monotonic covering route 39
4.2 SRG-v: Risoluzione del gioco 43
4.2.1 SRG-cycle: Risoluzione del gioco 45
4.3 PG: Enumerazione possibili covering cycles 46
4.3.1 Algoritmo esatto: enumerazione completa 46
4.3.2 Euristica statica: diametro massimo del sottografo . . 48
4.3.3 Euristica statica: ordinamento dei target per valore . . 51
4.3.4 Euristica dinamica: strategia dell'attaccante 54
4.4 PG: Ricerca covering cycle 57
4.4.1 Algoritmo esatto: backtracking con forward checking . 58

5 Analisi sperimentale **63**
5.1 Setting sperimentale 63
5.2 Esperimento: ricerca covering cycles 64
5.2.1 Risultati: ricerca covering cycles 65
5.3 Esperimento: risoluzione esatta del gioco 67
5.3.1 Risultati: risoluzione esatta del gioco 68
5.4 Esperimento: risoluzione approssimata del gioco 72
5.4.1 Risultati: risoluzione approssimata del gioco 76

6 Conclusioni e sviluppi futuri **85**
6.1 Conclusioni 85
6.2 Sviluppi futuri 86

Bibliografia **89**

Elenco delle figure

2.1 Esempio di rappresentazione ad albero per i giochi in forma estesa. 10
2.2 Rappresentazione ad albero per il gioco dei politici. 12
2.3 Rappresentazione ad albero per il gioco dei politici: ridotto . 12

3.1 Albero di gioco, in cui assumiamo v come vertice corrente per $\mathcal{D}$. r è una sequenza di vertici chiamata *rotta*. 29

4.1 Esempio di grafo da pattugliare 38

5.1 Covering cycles trovati su istanze di densità pari a 0.25 a seconda del bound k scelto. 65
5.2 Covering cycles trovati su istanze di densità pari a 0.75 a seconda del bound k scelto. 66
5.3 Andamento del valore esatto del gioco a seconda dell'incertezza α, mediato sui grafi con densità del 25%. 70
5.4 Andamento del valore esatto del gioco a seconda dell'incertezza α, mediato sui grafi con densità del 75%. 71
5.5 Andamento della cardinalità della rotta migliore, calcolata in modo esatto, a seconda dell'incertezza α, mediato sui grafi con densità del 25%. 73
5.6 Andamento della cardinalità della rotta migliore, calcolata in modo esatto, a seconda dell'incertezza α, mediato sui grafi con densità del 75%. 74
5.7 Confronto delle euristiche, a seconda dell'incertezza α, mediato sui grafi con densità del 25%, utilizzando 0 permutazioni nella creazione approssimata dei covset 78
5.8 Confronto delle euristiche, a seconda dell'incertezza α, mediato sui grafi con densità del 75%, utilizzando 0 permutazioni nella creazione approssimata dei covset 79

5.9 Confronto delle euristiche, a seconda dell'incertezza α, mediato sui grafi con densità del 25%, utilizzando 10 permutazioni nella creazione approssimata dei covset. 80
5.10 Confronto delle euristiche, a seconda dell'incertezza α, mediato sui grafi con densità del 75%, utilizzando 10 permutazioni nella creazione approssimata dei covset. 81
5.11 Confronto delle euristiche, a seconda dell'incertezza α, mediato sui grafi con densità del 25%, utilizzando 20 permutazioni nella creazione approssimata dei covset. 82
5.12 Confronto delle euristiche, a seconda dell'incertezza α, mediato sui grafi con densità del 75%, utilizzando 20 permutazioni nella creazione approssimata dei covset. 83

Elenco delle tabelle

2.1 Esempio bidimensionale della matrice M. 13
2.2 Bimatrice del dilemma del prigioniero. 14
2.3 Bimatrice del gioco degli appuntamenti. 15
2.4 Bimatrice di un gioco a somma zero. 17

3.1 Bimatrice di un PSG, data la rotta R. 33

5.1 Percentuale di grafi il cui preprocessing ha ecceduto il tempo limite. 69

Capitolo 1

Introduzione

"Chi osa interrompere il sonno di Pdor, figlio di Kmer, della tribù di Ishtar, della terra desolata dei Kfnir, uno degli ultimi sette saggi: Pfulur, Galér, Astaparigna, Sùsar, Param, Fusus e Tarìm, eh?"

Giovanni (Pdor), Tel chi el telun

In questo capitolo verranno presentate le aree di studio toccate dalla ricerca svolta, i contributi dell'autore a suddetta ricerca e verrà infine descritta la struttura della tesi.

1.1 Area di ricerca

Supponiamo di avere un quartiere finanziario da proteggere. Nel quartiere ci sono varie banche, di diversa importanza e con diversi sistemi di sicurezza. Un ladro, a seconda delle contromisure, impiega un certo tempo per entrare e ripulire il caveau. Avendo a disposizione una sola unità di vigilanza, qual è il percorso migliore di pattugliamento? Ovvero, qual è il percorso che, in caso di furto, mi garantisce mediamente la minor perdita economica?

Questo problema, più o meno verosimile, rientra sotto l'ala dei Patrolling Security Games (detti PSG), una particolare sottocategoria dei giochi non cooperativi, descritti dalla teoria dei giochi. Poichè il nostro "quartiere" verrà modellato con un grafo, le nostre banche con dei nodi e il ladro e l'unità di vigilanza come agenti, suddetto problema rientra anche nella categoria dei sistemi multiagente, nonchè dell'intelligenza artificiale.

1.2 Contributi

Visto che il problema è noto e ben studiato, quale valore aggiunto porta la nostra ricerca? E' presto detto. Nel nostro scenario in Sezione 1.1, abbiamo supposto che il ladro possa eludere tutti i sistemi di sorveglianza (mettendoci più o meno tempo, a seconda della loro complessità), e dunque che l'unico modo per essere catturato durante un assalto sia che la pattuglia passi di lì per caso prima che abbia terminato il lavoro. Questo è, ovviamente, irrealistico.

Introduciamo quindi dei rilevatori che, supponiamo, il ladro non riesca o non possa disattivare. E che, se attivati, diano l'allarme all'unità di pattugliamento. Per essere verosimile, però, il modello di questi sensori deve includere un certo margine di incertezza, ovvero un particolare parametro α che indica la percentuale di *falso negativo*, ovvero la probabilità che, pur avvenendo un furto, l'allarme non scatti. Inoltre, per minimizzare il numero di sensori nel quartiere, in quanto molto costosi, ogni sensore copre una certa area del quartiere, e non una sola banca.

La nostra ricerca porta quindi a valutare la miglior strategia di pattugliamento al variare dell'importanza degli obiettivi (quanti soldi ci sono nel caveau?), delle misure di sicurezza (quanto tempo impiega il ladro a portare a termine il furto?) e, soprattutto, al variare dell'incertezza dei sensori (quanto è probabile che non suoni l'allarme, nonostante ci sia un furto in atto?).

Come vedremo in seguito, si può dimostrare che il problema di trovare una strategia di pattugliamento ottimo è un problema NP-HARD (non solo, anche trovare un percorso di pattugliamento che "protegga" solo determinati obiettivi è NP-HARD), dunque vedremo come risolverlo in modo esatto su grafi "piccoli", e come risolverlo in modo euristico su grafi "grandi", creando e testando euristiche per la creazione degli insiemi di obiettivi da proteggere, per la ricerca di una rotta che li copra e per la risoluzione del gioco stesso.

1.3 Struttura della tesi

La tesi è strutturata nel modo seguente.

Nella Sezione 2 viene descritto lo stato dell'arte riguardo al problema di cui ci stiamo occupando, dando un minimo di background necessario per capire

i concetti e le tecniche utilizzate per risolverlo.

Nella Sezione 3 viene descritto formalmente il problema, vengono definiti i vari parametri utilizzati per la descrizione dei grafi, delle rotte e delle soluzioni del gioco. Viene inoltre mostrata la NP-HARDNESS del problema.

Nella Sezione 4 vengono descritti tutti gli algoritmi utilizzati per risolvere in modo esatto, o approssimato, i problemi di ricerca degli insiemi di obiettivi, delle rotte di pattugliamento, e di risoluzione del gioco.

Nella Sezione 5 vengono descritti i setting per le analisi, l'esecuzione delle analisi stesse, e i risultati ottenuti, in base a valore del gioco, tempo di esecuzione e, se possibile, rapporto tra valore approssimato e valore esatto.

Nella Sezione 6 vengono esposte le conclusioni della nostra ricerca e le possibili analisi future.

Capitolo 2

Stato dell'arte

> *"Giovanni: Ma cosa ne vuoi capire tu Giacomo, non sai neanche cos'è una pialla!*
> *Giacomo: Ma cosa c'entra? Non è che uno deve costruire i mobili per saperli apprezzare!*
> *Giovanni: No caro: chi sa fare, sa capire!*
>
> Aldo,Giovanni e Giacomo, Chiedimi se sono felice

In questa sezione verranno presentati tutti i concetti e le definizioni necessarie per comprendere appieno quanto è stato fatto nell'ambito della nostra ricerca. Nello specifico, verranno date delle nozioni di teoria dei giochi (e in particolare, dei *Patrolling Security Games*) e di complessità algoritmica. Infine, verranno presentate le ultime ricerche svolte in questo campo, da cui la nostra ricerca ha preso spunto.

2.1 Teoria dei giochi

Poichè in seguito modelleremo il nostro problema come un "gioco", ovvero un particolare modello studiato dalla Teoria dei Giochi [25], in questa sezione verranno date definizioni e accenni ai concetti necessari per comprendere il nostro modello.

2.1.1 Cos'è un gioco?

Definizione 1 *Un gioco è un particolare modello di interazione fra agenti composto da:*

- $N = 1, 2, \ldots, n$, *ovvero l'insieme degli agenti o, più correttamente, dei giocatori.* [1]

- $A = \{A_1, A_2, \ldots, A_n\}$, *ovvero l'insieme delle possibili azioni compibili dai vari agenti. L'insieme* A_i, *ovviamente, rappresenta le azioni possibili per l'agente* i.

- X, *l'insieme dei possibili esiti del gioco.*

- $f : A_1 \times A_2 \times \cdots \times A_n \to X$, *la funzione degli esiti, ovvero la funzione che associa ad una particolare sequenza o insieme di azioni, un esito corrispondente.*

- U_i, *ovvero l'utilità del gioco per il giocatore i. In pratica, quanto sia conveniente al giocatore i decidere di partecipare al gioco.* [2]

- $u_i : X \to \mathbb{R}$, *ovvero la funzione di utilità che associa ad ogni esito del gioco un'utilità per il giocatore i.*

Purtroppo la sola descrizione del gioco non è sufficiente per iniziare la modellazione. Non si hanno infatti informazioni sui giocatori e sulla loro metodologia di gioco (un giocatore potrebbe, ad esempio, giocare randomicamente, o ignorare completamente le mosse fatte da un suo avversario). Noi quindi assumeremo che:

- Il giocatore è egoista, ovvero cercherà sempre di giocare la sequenze di azioni che gli garantiscono la maggiore utilità possibile. Per capire meglio, citiamo un famoso esempio [20]. Un uomo molto ricco, per testare l'amicizia tra 2 persone, propose loro un gioco. Chiese al primo giocatore di scegliere tra 2 opzioni: l'opzione A avrebbe garantito un milione di dollari a testa per entrambi, mentre l'opzione B avrebbe dato un milione e un dollaro al primo giocatore, e nulla al secondo. Poichè abbiamo assunto che i giocatori siano egoisti, il primo giocatore sceglierà sempre l'opzione B, indipendentemente da quanto venga dato al secondo giocatore.[3] Trovare la soluzione di un gioco significa quindi, per noi, fornire ad ogni giocatore la serie di azioni che gli garantisca di massimizzare la propria utilità.

[1] In realtà N è un qualsiasi insieme di label che identifichino gli agenti. Esempi di possibili N sono $N = \{1, 2, 3\}$, $N = \{A, B\}$,$N = \{Nicola, Giuseppe\}$.

[2] Nella nostra analisi non verrà utilizzata, ma viene lasciata per correttezza formale.

[3] Esistono vari modi per modellare l'utilità. Qui abbiamo supposto che l'utilità del primo giocatore fosse legata alla quantità di soldi guadagnati, ma ovviamente avremmo potuto tenere conto degli aspetti sociali e psicologici della cosa, garantendo un' utilità maggiore per l'opzione A.

- Il giocatore è razionale. Il concetto di "razionale" è, purtroppo, molto lasco, e oggetto di numerose discussioni [13]. Per le nostre analisi, e per la teoria dei giochi in generale, il giocatore viene considerato razionale se:

  1. è capace di ordinare gli esiti del gioco in base a una particolare preferenza (generalmente fatto tramite la funzione di utilità)
  2. è capace di analizzare completamente il problema, ovvero è capace di analizzare tutte le conseguenze delle proprie azioni e delle azioni giocate dagli avversari, in modo ricorsivo. Per chiarire meglio, proponiamo un altro esempio [20]. Supponiamo ci siano 100 persone in una stanza, a cui viene chiesto di scrivere su un foglietto un numero tra 1 e 100. Chi si avvicinerà di più alla metà della media dei numeri scritti, vincerà un premio cospicuo. In caso di pareggio, il premio verrà ovviamente diviso tra i vincitori. Essendo capace di analizzare il gioco, il giocatore sa che dovrà scrivere un numero tra 1 e 50, perchè la metà della media non sarà mai oltre la metà del numero massimo. Ma sa anche che anche gli altri giocatori sanno questa cosa, e quindi dovrà scrivere un numero tra 1 e 25 (la metà della metà). Ma anche gli altri giocatori lo sanno, e così via. La soluzione razionale del gioco, benchè assai improbabile in realtà, in quanto difficilmente dei giocatori umani giocano razionalmente, è che tutti i giocatori scrivano 1, vincendo tutti a pari merito.

Ora possiamo finalmente analizzare i vari tipi di gioco. Ne esistono molti, generalmente divisi in base alla strategia di risoluzione, alla rappresentazione oppure alla quantità di informazioni dei giocatori sul gioco [19]. Noi ci focalizzeremo principalmente sui giochi *non-cooperativi* [21], ovvero giochi in cui gli agenti competono tra loro, *ad informazione perfetta* [28], ovvero giochi nel quale tutto è osservabile, e *deterministici*, ovvero giochi nei quali ogni sequenza di azioni porta con certezza ad un certo esito.

2.1.2 Giochi in forma estesa

Formalmente, la forma estesa [17] di un gioco è una semplice rappresentazione matematica di tutte le informazioni rilevanti per il gioco stesso, ovvero:

- lo stato iniziale del gioco,

- tutte le possibili evoluzioni del gioco,
- tutti i possibili esiti del gioco, e le preferenze dei giocatori al riguardo.

Generalmente, tutte queste informazioni vengono raccolte in un albero come quello in Figura 2.1. In particolare:

- i nodi rappresentano le decisioni dei giocatori. Generalmente sono ordinati in modo sequenziale (il primo giocatore è quello alla radice dell'albero, e via discendendo), ma esistono casi particolari in cui l'informazione è imperfetta e dunque, benchè i giocatori siano ordinati in un albero, in realtà giocano contemporaneamente[4]. Nell'esempio, i nodi vengono marcati con la coppia $i.j$, dove i rappresenta il giocatore e j il nodo;
- i rami indicano le varie azioni possibili dei giocatori;
- le foglie indicano le utilità dei giocatori per quel particolare esito. In particolare u_{ij} indica l'utilità del giocatore j rispetto all'esito i.

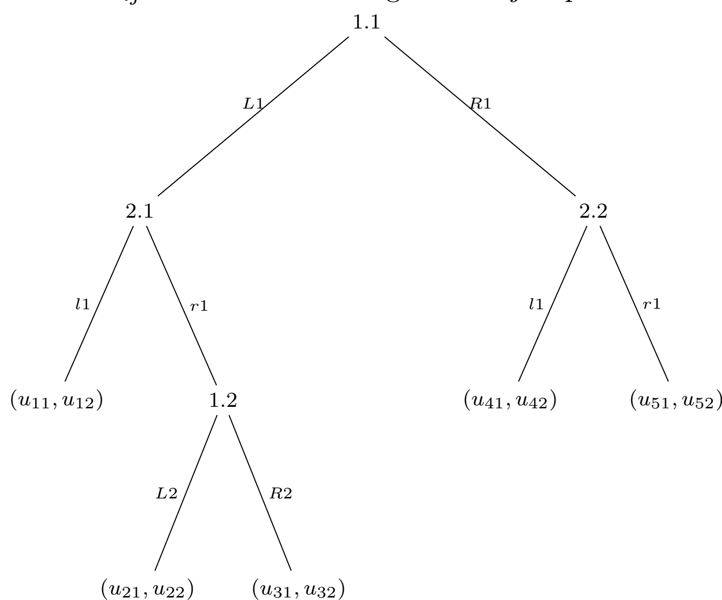


Figura 2.1: Esempio di rappresentazione ad albero per i giochi in forma estesa.

Per risolvere un gioco in forma estesa si può utilizzare una potente tecnica chiamata *backward induction*, o *induzione a ritroso*. Tale tecnica può però

[4]Per indicare che un giocatore non ha informazione sul nodo decisionale in cui si trova, si utilizza spesso una linea tratteggiata tra i nodi decisione "equipossibili". Comunque, non verrà qui trattato in quanto non necessario alla nostra ricerca.

essere utilizzata solo se l'informazione del gioco è completa [4], in quanto ogni giocatore deve poter analizzare il gioco in modo completo (l'assioma di razionalità che abbiamo assunto), e se l'albero è relativamente ristretto[5]. La tecnica è semplice: si analizza l'albero dalle foglie, e si procede a ritroso fino alla radice. Per meglio dire, a ogni iterazione:

- controllo chi è il giocatore i incaricato della decisione (ovvero, controllo il nodo superiore, se esiste). Dopo di che, ordino le foglie in base all'utilità del giocatore i. Ovviamente, il giocatore i sceglierà la foglia con utilità più alta per lui (in caso di pareggio, l'albero si sdoppia e si valutano entrambi i casi);
- sostituisco il nodo superiore con la foglia scelta dal giocatore i;
- ripeto fino ad arrivare alla radice dell'albero.

Per chiarire meglio, introduciamo un esempio, tratto da [20]. Supponiamo che ci siano due politici che debbano prendere una decisione necessaria ma impopolare pur di salvare il proprio paese (ad esempio, visto i recenti avvenimenti, aumentare le tasse per rimpinguare le casse statali). Supponiamo che basti l'approvazione di uno solo dei due affinchè le misure vengano messe in atto, e supponiamo che la decisione sia pubblica. Ovviamente, nessuno dei due politici vuole attirare su di sè l'odio del popolo, ma al tempo stesso nessuno dei due può permettersi di portare il paese alla rovina per non aver preso la decisione. Per un qualche privilegio sociale, il primo politico voterà prima del secondo. Il gioco può quindi essere descritto con il seguente albero (Figura 2.2). I due politici sono indicati con le label $\{1, 2\}$, e le due azioni "approva, rifiuta" con le label $\{y(es), n(o)\}$. L'utilità di ogni politico è:

$$u_i = \begin{cases} 2, & \text{se la proposta viene approvata, ma lui si è opposto} \\ 1, & \text{se la proposta viene approvata, ma lui non si è opposto} \\ 0, & \text{se la proposta non viene approvata} \end{cases}$$

Procediamo quindi alla risoluzione del gioco per *backward induction*. Al nodo 2.1, il giocatore 1 ha già approvato la proposta. Non ha dunque senso per il secondo giocatore approvarla di nuovo, attirandosi così l'odio della gente. Deciderà quindi di opporsi alla riforma (tutto questo è matematicamente riportato tramite le utilità, infatti l'utilità della foglia n è maggiore

[5] Il gioco degli scacchi è infatti un gioco ad informazione completa ed è analizzabile in forma estesa, ma con un ordine di 10^{120} stati.[24]

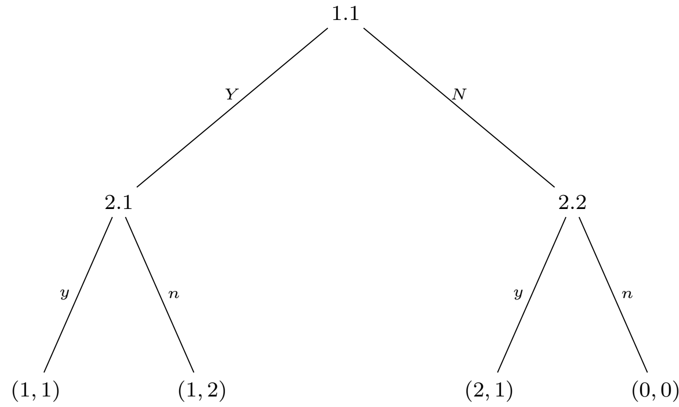


Figura 2.2: Rappresentazione ad albero per il gioco dei politici.

dell'utilità della foglia y). Sostituiamo quindi il nodo 2.1 con la foglia (1,2). Ripetiamo lo stesso procedimento per il nodo 2.2. Stavolta il primo giocatore si è già opposto alla riforma e quindi, pur di non far fallire il paese, il secondo giocatore è "costretto" ad approvarla, garantendosi l'odio del popolo (ma meglio odiati, che in rovina, come dimostrano le utilità nelle due foglie). Sostituiamo quindi il nodo 2.2 con la foglia (2,1), ottenendo così un nuovo albero ridotto (vedi Figura 2.3).

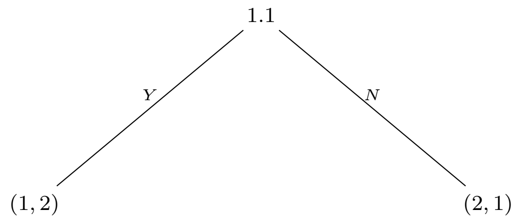


Figura 2.3: Rappresentazione ad albero per il gioco dei politici: ridotto

Riapplicando lo stesso procedimento per il nodo 1.1, che è la radice e dunque l'ultima iterazione possibile dell'algoritmo, si vede come al primo giocatore convenga opporsi alla riforma, in quanto già sa (ecco di nuovo l'ipotesi di razionalità) che il secondo giocatore, pur di non far fallire il paese, approverà la riforma. L'esito del gioco è quindi (2,1), con il primo giocatore che si oppone ed e il secondo che approva la riforma.

Anche il gioco che andremo ad analizzare si può descrivere in questo modo, ma esiste un modo più semplice e compatto, che presenteremo nella sezione seguente.

2.1.3 Giochi in forma strategica

Mentre la rappresentazione in forma estesa è più utile per giochi nei quali esiste una sequenzialità (es. dei turni), e nei quali vi siano più interazioni tra i giocatori (più di un turno o di una mano, ad esempio), la rappresentazione in forma strategica [28] è più adatta a quei giochi a un turno solo, in cui entrambi i giocatori fanno la propria mossa contemporaneamente (pari e dispari, carta, forbice, sasso [18] e così via...). Generalmente viene usata una matrice n-dimensionale M, dove n è il numero dei giocatori, ove l'elemento $m_{i_1,i_2,\ldots,i_n}$ è una n-pla rappresentante l'utilità dei giocatori, giocando le azioni $i_1, i_2, \ldots, i_n$.

		Giocatore 2	
		$A_{2,1}$	$A_{2,2}$
Giocatore 1	$A_{1,1}$	(u_{11}, u_{12})	(u_{21}, u_{22})
	$A_{1,2}$	(u_{31}, u_{32})	(u_{41}, u_{42})

Tabella 2.1: Esempio bidimensionale della matrice M.

Prima di spiegare come risolvere questo tipo di gioco, ci servono ancora un paio di definizioni.

Definizione 2 *La strategia $\underline{\sigma_i}$ è un vettore che indica la probabilità con la quale il giocatore i giocherà le proprie azioni. Ovviamente, deve rispettare le classiche regole di una funzione di probabilità, ovvero:*

- $\forall a \in A_i \quad \sigma_{i,a} \geq 0$,
- $\sum_{\forall a \in A_i} \sigma_{i,a} = 1$.

Definizione 3 *Una strategia $\underline{\sigma_i}$ si dice*

- *pura, se $\exists a \in A_i$ tale che $\sigma_{i,a} = 1$,*
- *mista, se non è pura,*
- *completamente mista, se $\forall a \in A_i, \sigma_{i,a} > 0$.*

Gli esempi seguenti saranno quasi tutti in strategie pure per semplicità.

2.1.4 Risoluzione di giochi in forma strategica

Esistono diversi modi per risolvere un gioco in forma strategica. Di seguito descriveremo quelli necessari ai fini della nostra ricerca.

2.1.4.1 Eliminazione delle strategie dominate

Definizione 4 *Si chiamino x e y le strategie di due giocatori, e $f(x,y)$ e $g(x,y)$ due funzioni che assegnino rispettivamente l'utilità del primo e del secondo giocatore se giocano le strategie x e y. Una strategia x è dominata da una strategia z, se per ogni possibile strategia avversaria y si ha che $f(x,y) < f(z,y)$.*

Uno dei modi per risolvere i giochi in forma strategica, è eliminare ricorsivamente le strategie dominate [14]. Intuitivamente, infatti, un giocatore egoista giocherà sempre ciò che gli garantirà un'utilità più elevata, e dunque tralascerà le strategie dominate.

Un famoso e abusato esempio è il dilemma del prigioniero [26]. Due persone sono fortemente sospettate di aver compiuto un crimine, ma le prove indiziarie non permettono alla polizia di poterli incriminare con certezza. Serve dunque una confessione. Per evitare che i due si mettano d'accordo, vengono imprigionati in due celle separate, e a entrambi viene proposto lo stesso patto. Se confessano entrambi, verrà loro applicata una pena ridotta, e dunque sconterebbero solo 5 anni di carcere. Se solo uno dei 2 confessa, chi avrà confessato sarà libero grazie alla sua collaborazione, mentre l'altro sconterà 7 anni di carcere. Se nessuno dei due confessa, verranno incriminati con le prove raccolte per un reato minore, e sconteranno solo 1 anno di carcere a testa.

Come modellarlo in termini di un gioco? Le azioni per entrambi i giocatori sono {confessa, non confessa}, mentre le utilità degli esiti sono semplicemente gli anni da scontare in carcere (ovviamente in negativo, poichè più sono, e meno è invitante per il giocatore). Otteniamo così la Tabella 2.2.

	Confessa	Non confessa
Confessa	$(-5,-5)$	$(0,-7)$
Non confessa	$(-7,0)$	$(-1,-1)$

Tabella 2.2: Bimatrice del dilemma del prigioniero.

E' abbastanza evidente che, potendosi mettere d'accordo e fidandosi l'un l'altro, entrambi i giocatori non confesserebbero. Ma in questo caso, i giocatori non possono accordarsi, e giocano dunque in maniera egoistica. Cosa conviene fare, dunque? Controlliamo ad esempio le righe, ovvero le azioni del primo giocatore. Se il giocatore confessa, ha in entrambi casi un'utilità

più alta che non confessando ($-5 > -7$ e $0 > -1$). Infatti, non sapendo che cosa abbia fatto l'altro giocatore, non ci conviene rischiare. Se io confesso, e lui ha confessato, mi risparmio 2 anni di prigione. Se io confesso, e lui non ha confessato, io vengo addirittura liberato. Il problema è che questo vale anche per il secondo giocatore. Eliminando così la seconda riga e la seconda colonna, otteniamo che la risoluzione del gioco è confessare per entrambi i prigionieri, garantendosi così 5 anni di carcere entrambi. Ma cosa succede se non esistono strategie dominate o dominanti?

2.1.4.2 Equilibri di Nash

Un giovane John Nash, insieme ai suoi amici al bar, nota una bellissima bionda insieme alle sue amiche. La ragazza è così bella che attira ovviamente gli sguardi e gli interessi di tutti. Ma è anche ovvio che, se tutti ci provassero solo con lei, tutti, tranne forse un gran fortunato, tornerebbero a casa con le pive nel sacco. E una volta rifiutati dalla bionda, non si può certo provarci con le sue amiche, che si offenderebbero per essere state una "seconda scelta".

Proviamo quindi a modellare questo interessante gioco. Per semplicità, supponiamo che ci siano solo 2 giocatori, e che entrambi abbiamo due possibili azioni: provarci con la bionda, oppure provarci con una delle sue amiche. Le utilità vengono da sè. Se entrambi ci provano con la bionda, ovviamente si ostacolano a vicenda, e nessuno dei due riuscirà a concludere. Se entrambi ci provano con una delle amiche, ovviamente non la stessa per entrambi, non si ostacolano, e riescono sicuramente a conquistarla. Se un solo giocatore ci prova con la bionda, supponiamo per semplicità che riesca a conquistarla. Otteniamo quindi il gioco in Tabella 2.3.

	Amica	Bionda
Amica	$(1, 1)$	$(1, 2)$
Bionda	$(2, 1)$	$(0, 0)$

Tabella 2.3: Bimatrice del gioco degli appuntamenti.

Come possiamo notare, non esistono strategie dominate. Andando con un'amica, infatti, il giocatore ci guadagna rispetto alla scelta di andare entrambi con la bionda ($1 > 0$), ma ci perde rispetto al caso in cui solo lui abbia scelto di provarci con la bionda ($1 < 2$). Come risolvere il gioco, quindi? Chiunque abbia visto il film *A Beautiful Mind* [1] sa che la famosa soluzione proposta dal giovane Nash, chiamata appunto *equilibrio di Nash*

[22], è quella di andare solo con le amiche, tralasciando la bionda. Ma solo chiunque abbia studiato un po' di teoria dei giochi, sa che questo è uno sbaglio clamoroso dello sceneggiatore. Infatti:

Definizione 5 *Chiamati X, Y gli insiemi delle possibili strategie del primo e secondo giocatore, f e g le funzioni che assegnano loro le utilità rispetto alle strategie giocate, la coppia $(\bar{x}, \bar{y})$ è un equilibrio di Nash se e solo se*

- $f(\bar{x}, \bar{y}) \geq f(x, \bar{y}) \quad \forall x \in X$
- $g(\bar{x}, \bar{y}) \geq g(\bar{x}, y) \quad \forall y \in Y$

Questa definizione un po' involuta si può spiegare a parole come "se il giocatore sa che l'avversario farà quella mossa, allora la sua risposta migliore sarà questa". Ovviamente, da parte di entrambi i giocatori. Cerchiamo di chiarire con l'esempio. Supponiamo di giocare nei panni del primo giocatore. Non sapendo cosa fare, immagina tutti i casi possibili. Se il secondo giocatore ci proverà con un'amica, allora tanto vale provarci con la bionda (sulla colonna amica, $2 > 1$). Mentre se ci prova con la bionda, tanto vale provarci con l'amica (sulla colonna bionda, $1 > 0$). Lo stesso, scambiando righe e colonne, per il secondo giocatore. Si individuano così due equilibri di Nash, che sono le strategie (Bionda, Amica) e (Amica, Bionda). La vera soluzione del gioco è, quindi, accordarsi prima con gli amici per chi ci debba provare con la bionda, e provarci quindi con le sue amiche.

2.1.4.3 Giochi a somma zero e minmax

Gli equilibri di Nash, come abbiamo visto, sono però abbastanza complessi da trovare. Esiste però una particolare categoria di giochi, nei quali si può trovare facilmente un particolare equilibrio di Nash, chiamato *minmax*: i giochi a somma zero [28].[6]

Definizione 6 *Un gioco si dice a somma zero quando, chiamato W l'insieme degli esiti, $u_i(w)$ l'utilità del giocatore i per l'esito w, N l'insieme dei giocatori, si ha*

$$\sum_{i \in N} u_i(w) = 0 \quad \forall w \in W$$

[6] I concetti presentati in seguito valgono in generale per tutti i giochi a somma costante, che sono riconducibili ai giochi a somma zero tramite una trasformazione affine.

Nel caso a due giocatori i, j, è evidente che $u_i = -u_j$. E' dunque inutile scrivere due volte le utilità, basta scriverle per un solo giocatore (ad esempio, il primo), e poi trovare la soluzione di conseguenza. Ma come si risolve questo tipo di gioco? Andiamo per intuizione. Il primo giocatore cercherà di guadagnare la maggiore utilità possibile. Il secondo giocatore, poichè la sua utilità è l'opposto di quella del primo giocatore, deve invece fare in modo che quest'ultimo guadagni la minore utilità possibile. E' quindi evidente che il primo giocatore, per effetto delle scelte del secondo, guadagnerà il massimo dei minimi sulle righe delle matrice, mentre il secondo, per effetto del primo, guadagnerà il minimo dei massimi sulle colonne. Quindi:

Definizione 7 *Una coppia di strategie $(\bar{x}, \bar{y})$ è una soluzione minmax per un gioco a somma zero se*

$$f(x, \bar{y}) \leq f(\bar{x}, \bar{y}) \leq f(\bar{x}, y) \quad \forall x \in X, y \in Y$$

Chiariamo con un esempio, sempre tratto da [20]. Prendiamo il gioco in Tabella 2.4, di cui purtroppo non conosciamo le regole reali.

	A	B	C
A	4	3	1
B	7	5	8
C	8	2	0

Tabella 2.4: Bimatrice di un gioco a somma zero.

Il primo giocatore cercherà di massimizzare la propria utilità. Ma per effetto del secondo giocatore, riuscirà al massimo a prendere il minimo sulle righe. Ovvero, supponiamo che il primo giocatore giochi C, cercando di prendersi un'utilità di 8. Il secondo giocatore, di certo, si opporrà, e quindi giocherà anche lui C, portando al primo giocatore una bassissima utilità, pari a 0. Lo stesso per ogni riga. Le possibili utilità del primo giocatore sono quindi $\{1, 5, 0\}$ tra cui, evidentemente, sceglierà l'utilità 5, ovvero l'azione B. Dal punto di vista del secondo giocatore è, invece, il contrario. Supponiamo che giochi l'azione C per cercare di guadagnare un'utilità di 0. E' evidente che il primo giocatore giocherà invece B, portando così l'utilità a 8. Lo stesso per ogni colonna. Le possibili utilità del secondo giocatore sono quindi $\{8, 5, 8\}$. Dovendo scegliere il più basso, sceglierà quindi B. La soluzione *minmax* è quindi (B,B), che garantisce un utilità pari a 5.

Noi modelleremo il nostro problema con un gioco a somma uno, per cui valgono comunque gli stessi concetti qui presentati.

2.2 Complessità Algoritmica

2.2.1 Qualche definizione

Prima di iniziare a descrivere la nostra ricerca, bisogna chiarire qualche termine [27] e alcune delle notazioni [8] che verranno utilizzate in seguito.

Definizione 8 *Con il termine "tempo di computazione" (oppure "complessità computazionale" o "complessità temporale") di un certo algoritmo per un certo problema, si intende la quantità di operazioni elementari necessarie all'algoritmo per risolvere suddetto problema.*

Si ricorda che le operazioni elementari considerate sono l'assegnamento di una variabile, una somma, una sottrazione, una moltiplicazione o una divisione (queste ultime in realtà sono le composizione di più operazioni semplici, ma per i nostri calcoli le considereremo come elementari).

Definizione 9 *Con il termine "complessità spaziale" di un certo algoritmo per un certo problema, si intende la quantità di blocchi di memoria (oppure di bit, a seconda del calcolo) necessarie all'algoritmo per risolvere suddetto problema.*

Si ricorda che un blocco di memoria è la quantità di memoria necessaria a contenere una variabile semplice (generalmente un intero a 16 o 32 bit). Non utilizzeremo il calcolo di memoria sui singoli bit per ragioni di semplicità.

Definizione 10 *Con il simbolo $O(n)$ si indica l'ordine della complessità temporale (o spaziale) di un algoritmo, rispetto a una variabile o alla dimensione dell'istanza del problema n. Un algoritmo si dice di ordine $O(g(n))$ se e solo se, chiamata la sua complessità temporale reale $f(n)$, si ha*

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c, \quad c \neq \infty$$

Questa definizione è molto importante, in quanto spesso è molto complesso calcolare la vera complessità temporale $f(n)$, mentre è molto semplice calcolarne il suo ordine di grandezza, come faremo infatti in seguito.

2.2.2 Esponenziale vs Polinomiale

Un'antica leggenda [12] narra che un re di un ricco e prospero regno, il quale di certo non doveva risolvere problemi comuni come l'affitto o il mutuo da pagare, passava tutte le sue giornate ad annoiarsi. Stufo di tutto questo, imbandì un concorso per l'invenzione di un nuovo e mirabolante gioco che potesse finalmente distoglierlo dal suo tedio, promettendo al vincitore qualunque cosa avesse desiderato. Un giorno arrivò alla sua corte un uomo con una strana tavola quadrata, divisa in 64 caselle, su cui dispose 32 pedine, divise in vari ruoli, a seconda delle regole secondo le quali potevano muoversi. In pratica, l'inventore degli scacchi. Il re rimase così colpito dal gioco da cimentarcisi per ore e ore. Soddisfatto, chiese all'inventore quale ricompensa volesse. L'inventore fece una richiesta a dir poco curiosa. Chiese che gli venisse dato del riso, la cui quantità doveva essere calcolata nel suddetto modo: un chicco per la prima casella della sua scacchiera, due per la successiva, quattro per quella dopo ancora, e così via. Il re, pensando di essersela cavata con poco, chiese di calcolare e portare all'inventore ciò che chiedeva. Il povero re, evidentemente, non sapeva assolutamente cosa fosse una crescita esponenziale. Sarebbero infatti serviti

$$1 + 2 + 4 + 8 + 16 + \ldots = 2^{64} - 1 = 1.844 \cdot 10^{19}$$

chicchi di riso per soddisfare la richiesta dell'inventore. A seconda delle versioni, l'inventore viene poi assunto a corte, data la sua intelligenza, garantendosi una rendita vitalizia, oppure ottiene in cambio il regno, o viene invece ucciso per essersi preso gioco del re. Sperando che nella realtà la storia abbia avuto un lieto fine, vogliamo invece porre l'attenzione sul calcolo della quantità di chicchi di riso che è, appunto, un esempio di crescita esponenziale.

Definizione 11 *Un algoritmo si dice costante, rispetto alla dimensione dell'istanza n, se ha una complessità $O(1)$.*

Definizione 12 *Un algoritmo si dice polinomiale, rispetto alla dimensione dell'istanza n, se ha una complessità $O(n^c)$, con $c \geq 1$, costante.*

Definizione 13 *Un algoritmo si dice esponenziale, rispetto alla dimensione dell'istanza n, se ha una complessità $O(c^n)$, con $c \geq 1$, costante.*

2.2.3 $\mathcal{P}$ vs $\mathcal{NP}$

Dopo aver introdotto il concetto di complessità per gli algoritmi, facciamo lo stesso anche con i problemi in sè. Prima di tutto questo, però, bisogna classificare i problemi in due grosse macrocategorie:

- **problemi decisionali**, ovvero problemi a risposta YES/NO;
- **problemi di ricerca**, ovvero problemi per i quali l'algoritmo deve fornire una soluzione, oppure notificare che suddetta soluzione non esiste.

A seconda della macrocategoria, i problemi vengono divisi in diverse classi di complessità [23]. Per i problemi decisionali (su cui ci focalizzeremo maggiormente, in quanto più utili per la nostra ricerca) si hanno le classi $\mathcal{P}$ e $\mathcal{NP}$, mentre per i problemi di ricerca si hanno le classi $\mathcal{FP}$ e $\mathcal{FNP}$. Di seguito, elenchiamo le varie definizioni:

- $\mathcal{P}$: è la classe dei problemi che possono essere risolti in tempo polinomiale, la cui risposta è del tipo YES/NO;
- $\mathcal{NP}$: è la classe dei problemi per i quali si può produrre un certificato YES in tempo polinomiale. Ovvero, data una possibile soluzione al nostro problema, possiamo verificare in tempo polinomiale che essa sia una soluzione valida.
- $\mathcal{FP}$: è la classe dei problemi la cui soluzione è producibile in tempo polinomiale;
- $\mathcal{FNP}$: è la classe dei problemi per i quali di può produrre un certificato YES in tempo polinomiale. Ovvero, data una possibile soluzione al nostro problema, possiamo verificare in tempo polinomiale che essa sia una soluzione valida.

Come già detto, noi ci focalizzeremo solo sui problemi decisionali, e quindi sulle classi $\mathcal{P}$ e $\mathcal{NP}$. Esistono però problemi "difficili", che rientrano in una particolare categoria chiamata $\mathcal{NP}$*-HARD*. Ma prima di descriverli, ci servono alcune definizioni.

Definizione 14 *Un problema decisionale P1 è riconducibile in tempo polinomiale a un altro problema decisionale P2 (P1 $\propto$ P2), se per ogni istanza I1 del problema P1 si può ricavare in tempo polinomiale un'istanza I2 del problema P2, dalla cui soluzione si può ricavare in tempo polinomiale una soluzione dell'istanza I1.*

In pratica, se P1 $\propto$ P2, allora P2 è difficile almeno quanto P1. E quindi, se P1 è risolvibile in tempo polinomiale, e P1 $\propto$ P2, allora anche P2 è risolvibile in tempo polinomiale. Più formalmente

$$P1 \in \mathcal{P} \land P1 \propto P2 \implies P2 \in \mathcal{P}$$

L'operazione di riconducibilità serve per poter definire un particolare sottoinsieme della categoria $\mathcal{NP}$, ovvero i problemi *$\mathcal{NP}$-COMPLETI*, ed un'altra categoria di problemi difficili, ovvero i problemi *$\mathcal{NP}$-HARD*.

Definizione 15 *Un problema P si dice* $\mathcal{NP}$-COMPLETO *[15] se*

- $P \in \mathcal{NP}$*;*
- $P' \propto P \quad \forall P' \neq P, P' \in \mathcal{NP}$ *(ovvero, ogni problema P' in $\mathcal{NP}$ è riconducibile a P in tempo polinomiale).*

Utilizzare questa definizione per dimostrare che un problema è *$\mathcal{NP}$-COMPLETO* è però, quantomeno, scomodo. Un modo più semplice è dimostrare che un qualsiasi problema *$\mathcal{NP}$-COMPLETO* sia riconducibile al nostro problema, per dimostrare che esso stesso sia *$\mathcal{NP}$-COMPLETO* [7]. Ovvero, utilizzando la proprietà transitiva, e chiamando P2 il nostro problema:

$$P \propto P1 \land P1 \propto P2 \implies P \propto P2 \quad \forall P \in \mathcal{NP}$$

Se già intuitivamente i problemi $\mathcal{NP}$ e *$\mathcal{NP}$-completi* sembrano più "difficili" degli altri, esiste un'ultima categoria ancora più complessa.

Definizione 16 *Un problema si dice* $\mathcal{NP}$-HARD *[15] se* $P' \propto P \quad \forall P' \neq P, P' \in \mathcal{NP}$ *(ovvero, ogni problema P' in $\mathcal{NP}$ è riconducibile a P in tempo polinomiale).*

Si noti che la definizione è simile a quella dei problemi *$\mathcal{NP}$-COMPLETI*, ma c'è una differenza fondamentale: il problema non deve per forza appartenere a $\mathcal{NP}$. Il che significa che produrre un certificato YES per una qualsiasi soluzione potrebbe richiedere un tempo non polinomiale, e quindi anche solo la verifica di una soluzione potrebbe essere computazionalmente impraticabile.

[7] Ovviamente, il primo problema a essere dimostrato *$\mathcal{NP}$-COMPLETO* è stato dimostrato in altro modo, più precisamente con il *Teorema di Cook-Levin [9]*

Mostreremo poi nella Sezione 3.1.5 che, purtroppo, una grossa parte della nostra ricerca verte su un problema $\mathcal{NP}$-*HARD*, motivo per il quale nella Sezione 4 mostreremo varie euristiche e algoritmi per risolvere il problema in modo approssimato.

2.2.4 Il problema del millennio

Questa sezione, in realtà, è solo un piccolo chiarimento. Nella nostra dissertazione abbiamo detto (o almeno fatto intendere) che i problemi $\mathcal{NP}$ siano ben più difficili dei problemi $\mathcal{P}$. Benchè nella pratica, attualmente, sia così, non vi è alcuna dimostrazione teorica che questo sia vero [5]. Il che significa che potrebbero esistere degli algoritmi polinomiali che permettano di risolvere i problemi $\mathcal{NP}$, ma che semplicemente non siano stati ancora scoperti.

Questa dimostrazione teorica è uno dei famosi *Millenium Prize Problems*, ovvero dei particolari e importanti problemi matematici ancora irrisolti, per la cui soluzione il *Clay Mathematics Institute of Cambridge, Massachusetts* ha posto in palio un milione di dollari [2]. Noi, comunque, non avendo ancora degli algoritmi polinomiali che risolvano dei problemi $\mathcal{NP}$, useremo la $\mathcal{NP}$-*HARDNESS* come indicatore della complessità del problema.

2.3 Patrolling Security Games

Ora che abbiamo qualche nozione in più sulla teoria dei giochi e sulla complessità computazionale, possiamo addentrarci a parlare della sottocategoria di giochi con la quale modelleremo il nostro problema: i Patrolling Security Games (PSG) [7]. Un PSG è definito da:

- due giocatori, chiamati *attaccante* e *difensore*, le cui azioni variano a seconda del particolare PSG;
- un grafo $G = (V, E)$, con V insieme di vertici ed E insieme di archi non orientati;
- $T \subseteq V$, insieme dei target da attaccare/difendere;
- $\pi(t) \in [0, 1]$, valore del target t,$\forall t \in T$;
- $d(t)$, deadline (ovvero il tempo necessario per l'attacco) del target t,$\forall t \in T$.

Spieghiamo più nel dettaglio con l'esempio fatto nella Sezione 1.1. Il nostro quartiere finanziario è modellato tramite un grafo, in cui ad esempio

i vertici sono gli incroci, gli archi le strade e i target le banche. Il valore delle banche $\pi(t)$ dipende dalla quantità di soldi presenti nel caveau, e viene normalizzato per ottenere un valore nell'intervallo $[0, 1]$, più semplice per descrivere poi le utilità dei giocatori. Le deadline $d(t)$ non sono altro che il tempo necessario al ladro per superare le difese della banca, e quindi fuggire con il denaro del caveau. Il difensore, quindi, nel suo giro di pattuglia, dovrebbe riuscire a passare per ogni target/banca almeno ogni $d(t)$ unità di tempo, altrimenti il ladro potrebbe, banalmente, attaccare appena la pattuglia è passata.

Come possiamo definire le utilità di questo gioco? Ovviamente, per il ladro la situazione migliore è derubare la banca più importante senza venire catturato. Di contro, la situazione migliore per il difensore è che nulla venga svaligiato, o perchè ha protetto tutto alla perfezione, o perchè è riuscito a catturare il ladro in tempo. Visto che le utilità sono contrapposte, possiamo modellare il gioco come un gioco a somma costante[8]. Le utilità per l'attaccante sono quindi:

- 0, se viene catturato, o se non riesce ad attaccare alcunchè;
- $\pi(t)$, se riesce ad attaccare il target t (ovvero, se il difensore non è passato per il target entro $d(t)$ unità di tempo).

Per converso, le utilità del difensore sono:

- 1, se cattura il ladro, o se protegge tutti i target alla perfezione;
- $1 - \pi(t)$, se un ladro è riuscito ad attaccare il target t (ovvero, se il difensore non è passato per il target entro $d(t)$ unità di tempo).

Ora conosciamo quasi tutto del gioco. Ma quali sono le possibili azioni dell'attaccante e del difensore? Nel modello più generico, l'attaccante può

- attaccare il target t;
- aspettare un turno.

Di contro, il difensore, al momento in un vertice v può scegliere se

- rimanere nel vertice v;

[8] Più precisamente è un modello *leader-follower* di Stackelberg [29] ma, come ben spiegato in [7], si può vedere come un gioco a somma costante. Riprenderemo meglio il discorso in Sezione 3.1.2

- spostarsi al vertice v', tramite un arco e che collega v e v'.

Come si può facilmente notare, il gioco può protrarsi all'infinito, e calcolarne la soluzione è, a meno di casi particolari e istanze molto piccole, computazionalmente difficile. Rimandiamo a [7] per una spiegazione più dettagliata e formale.

2.3.1 PSG: riduzione a strategie deterministiche

Per facilitare la risoluzione del gioco, ridurremo le azioni del difensore alla scelta di un particolare ciclo di pattuglia deterministico, che continuerà a giocare ripetutamente fino alla fine del gioco. Più formalmente

Definizione 17 *Un ciclo di pattuglia deterministico è una sequenza di vertici* $(v_1, v_2, \ldots, v_1), v_i \in V$ *in modo tale che ognuno sia collegato al successivo da un arco* $e \in E$*, seguiti ripetutamente dal difensore per tutta la durata del gioco.*

Anche qui, rimandiamo a [7] per una descrizione più dettagliata, e per una dimostrazione della $\mathcal{NP}$-completezza per il problema di trovare un ciclo di pattuglia deterministico che garantisca al difensore di proteggere tutti i target.

2.3.2 PSG: introduzione degli allarmi

Un'interessante estensione al normale PSG viene presentata nell'articolo [6]. Per rendere più realistico il modello, infatti, vengono introdotti dei segnali di allarme con incertezza spaziale. Cosa significa con incertezza spaziale? Facciamo un esempio. Supponiamo di dover monitorare un grosso recinto dove sono rinchiusi degli animali rari, per evitare possibili fughe. Il recinto è molto grande, quindi che sensori potremmo usare per monitorarlo al meglio? Ad esempio, dei sensori di movimento ad ampio raggio. Usandone un numero limitato, saremmo comunque in grado di scoprire eventuali fughe, al costo di dover esaminare l'area di azione del sensore che ha lanciato l'allarme.

Formalmente, introduciamo:

- $S = \{s_1, s_2, \ldots, s_n\}$, l'insieme dei segnali;
- $p(s|t)$ ovvero la probabilità che il segnale s venga lanciato se il target t è sotto attacco, definita per ogni segnale s e target t.

Descriveremo più dettagliatamente nella Sezione 3.1 cosa questo comporti, e come cambi il modello. La nostra ricerca riprenderà infatti i risultati riportati in [6], ma estenderà il modello aggiungendo al difensore la capacità di pattugliare e aggiungendo agli allarmi un'incertezza di rilevamento, ovvero una probabilità di falso negativo.

Capitolo 3

Formulazione del problema

> *"Marina: Cosa fate nella vita?*
> *Giovanni: Beh noi lavoriamo nella meccanica di precisione, tecnologie avanzate al servizio di progettazioni particolari e specifiche. Non lo so... Hardware... Cioè creiamo dei supporti che serviranno per progettare grosse situazioni, non so strumenti di precisione per una svolta futura magari della meccanica, non so se mi spiego...*
> **Marina scuote la testa**
> *Aldo: Sì, insomma, abbiamo un negozio di ferramenta. Cioè, non è che il negozio di ferramenta è il nostro, noi ci lavoriamo come commessi, come galoppini insomma"*
>
> Aldo, Giovanni e Giacomo, Tre uomini e una gamba

In questa Sezione verrà descritto il problema da noi affrontato. In particolare verrà data una formulazione formale del modello, le aggiunte portate al modello standard dei PSG e gli obiettivi che ci poniamo con la nostra ricerca.

3.1 Formulazione del modello

3.1.1 PSG: allarmi con falsi negativi

Riprendiamo quanto detto nella Sezione 2.3. Il nostro modello sarà un'estensione dei normali PSG, quindi sarà composto da:

- due giocatori, chiamati *attaccante* e *difensore*;
- un grafo $G = (V, E)$, con V insieme di vertici ed E insieme di archi non orientati;
- w_e, peso degli archi, $\forall e \in E$;

- $T \subseteq V$, insieme dei target da attaccare/difendere;
- $\pi(t) \in [0, 1]$, valore del target t,$\forall t \in T$;
- $d(t)$, deadline (ovvero il tempo necessario per l'attacco) del target t,$\forall t \in T$;

come i normali PSG, a cui però aggiungiamo

- $S = \{s_1, s_2, \ldots, s_n\}$, l'insieme dei segnali, il cui supporto (ovvero i target coperti da ogni segnale s_i) verrà indicato con $T(s_i)$, mentre il suo converso, ovvero l'insieme dei segnali che coprono il target t_i, verrà indicato con $S(t_i)$;
- $p(s|t)$ ovvero la probabilità che il segnale s venga lanciato se il target t è sotto attacco, definita per ogni segnale s e target t;
- α, ovvero una percentuale di falso negativo per i segnali di allarme.

Sebbene il modello sembri identico a quello presentato in Sezione 2.3.2, c'è un'importantissima introduzione: i falsi negativi. I sensori reali, infatti, non sono perfetti, anzi, ma sono affetti da falsi positivi (ovvero un allarme viene lanciato nonostante non ci sia in atto alcun attacco) e falsi negativi (l'allarme, nonostante ci sia un attacco, non viene lanciato). Nella nostra ricerca, ci siamo focalizzati sui falsi negativi.

3.1.2 PSG in forma estesa

Nella Sezione 2.3 abbiamo descritto solo sommariamente la forma estesa di un PSG e la sua riduzione a un gioco a somma costante, per mantenere più generale e comprensibile il discorso. Entriamo quindi più nel dettaglio.

Analizziamo il nostro gioco usando il classico albero della rappresentazione in forma estesa (vedi Figura 3.1[9]), cercando di trovare dei sottoalberi ricorrenti per poter trasformare il gioco originale in un insieme di sottogiochi più facilmente risolvibili o rappresentabili. Partendo dalla radice dell'albero, si ha:

- *Radice (o primo livello) dell'albero.* L'attaccante decide se aspettare un turno (azione descritta con Δ nella Figura) o attaccare un target $t_i \in T$.

[9] L'attaccante verrà denotato con il simbolo $\mathcal{A}$, il difensore con $\mathcal{D}$ e i nodi della natura, ovvero i nodi nei quali le situazioni avvengono con una certa probabilità p, indipendente dai giocatori, con $\mathcal{N}$.

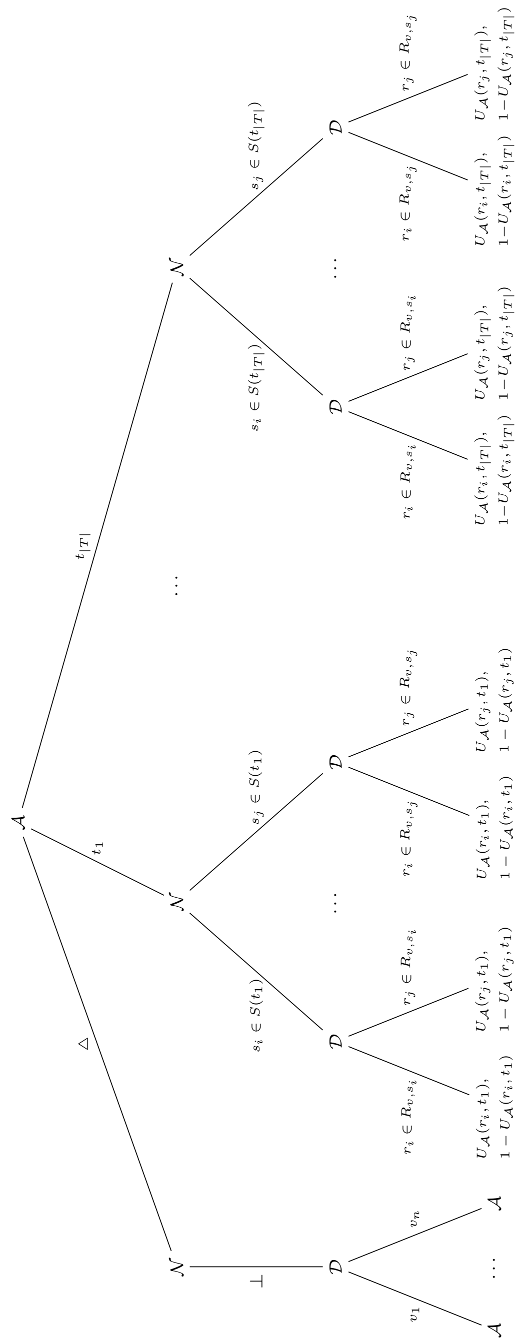


Figura 3.1: Albero di gioco, in cui assumiamo v come vertice corrente per $\mathcal{D}$. r è una sequenza di vertici chiamata rotta.

- *Secondo livello dell'albero.* $\mathcal{N}$ rappresenta la "decisione" probabilistica dell'allarme di suonare o non suonare, dato o non dato un attacco. Poichè abbiamo supposto che l'allarme non sia affetto da falsi positivi, abbiamo, indicando con $\perp$ il segnale nullo, ovvero un segnale che non fa scattare alcun allarme, che $p(\perp|\Delta) = 1$. In compenso, poichè abbiamo incluso i falsi negativi, abbiamo che, dato un attacco al target t_i:
  - l'allarme non suonerà con probabilità $p(\perp|i_i) = \alpha$;
  - l'allarme $s \in S(t_i)$ (ovvero un allarme che copre t_i) suonerà con probabilità $\tilde{p}(s|t_i) = (1 - \alpha)p(s|t_i)$.
- *Terzo livello dell'albero.* Il difensore decide, in base agli eventuali segnali d'allarme attivati, qual è il prossimo nodo da visitare tra quelli adiacenti al suo nodo attuale.
- *Quarto livello dell'albero.* Si hanno due casi:
  - se non c'è stato alcun attacco, l'albero si ripete esattamente nello stesso modo;
  - se c'è stato un attacco, l'attaccante è "bloccato" nell'attacco al nodo, e dunque non può più prendere decisioni. Il gioco passa totalmente nelle mani del difensore.

Dall'analisi abbiamo dunque ricavato che l'albero del PSG è fatto di sottoalberi ricorrenti, e quindi di sottogiochi risolvibili separatamente. Possiamo inoltre vedere che, a seconda delle azioni dell'attacante, abbiamo due sottogiochi (e quindi due sottoproblemi) diversi:

- *il target t_i è stato attaccato, e uno dei possibili segnali s che lo coprono è stato lanciato.* In questo caso, il gioco viene "bloccato" per l'attaccante, che impiega $d(t_i)$ per svaligiare/distruggere il target, mentre il difensore deve trovare la migliore risposta all'allarme, ovvero deve cercare di proteggere tutti i target appartenenti a $T(s)$, ovvero al supporto del segnale (ricordiamoci infatti che il segnale è spazialmente incerto, dunque il difensore non sa che l'attaccante abbia attaccato esattamente il target t_i). Chiamiamo questo gioco *Signal Response Game*, e poichè dipende dal vertice v nel quale si trova il difensore al momento dell'allarme, lo abbrevieremo in SRG-v.
- *non c'è stato alcun attacco, oppure l'allarme non è scattato.* Dunque il difensore continuerà a pattugliare...ma qual è il miglior percorso di pattuglia? Utilizzando la *backward induction*, come spiegato nella Sezione

2.1.2, possiamo ridurre l'albero con le soluzioni dei vari SRG-v, ottenendo così le utilità dei vari nodi da pattugliare. Chiameremo questo sottoproblema *Patrolling Game*, o più comodamente PG.

3.1.3 PG: best placement vs patrolling route

Ci sono due modi per risolvere un PG:

- *Best Placement.* Il difensore rimane fermo, e si muove solo in risposta a un segnale d'allarme. Un esempio possibile è il guardiano notturno di un museo, che esamina le telecamere di sorveglianza dalla sala di controllo, e si sposta solo se vede qualcosa di strano. In questo caso, il problema si riduce a trovare il miglior vertice v dove posizionare la sala di controllo, in modo tale che la risposta sia la più veloce possibile.

- *Patrolling Route.* Il difensore esegue una ronda predefinita fino a quando non riceve un segnale d'allarme. Utilizzando sempre l'esempio del museo, il guardiano circola fra le sale secondo un percorso predefinito, finchè non scatta un allarme, a cui poi dovrà rispondere nel minor tempo possibile. In questo caso il problema è più complesso, in quanto le rotte possibili sono infinite. Per semplicità, noi ci concentreremo esclusivamente sui *Covering Cycle*, ovvero particolari covering route nei quali il primo vertice e l'ultimo coincidono. Definiremo le covering route nella sezione successiva.[10]

3.1.4 SRG-v: risoluzione in forma strategica

Abbiamo quindi dimostrato che il nostro PSG è scomponibile in vari SRG-v, che possiamo, per semplicità, risolvere in forma strategica.

Come possiamo strutturare il nostro gioco? I due giocatori sono ovviamente il difensore e l'attaccante. Le azioni dell'attaccante sono, ovviamente, la scelta del target t_i da attaccare al momento in cui il difensore si trova nel vertice v. Di contro, il difensore deve scegliere un percorso per cercare di proteggere tutti i nodi appartenenti a $T(s)$ dove s è l'eventuale allarme lanciato dall'attacco del target t_i. Ma poichè esistono infiniti percorsi, quali conviene scegliere?

[10] Si noti che il best placement è un banale sottocaso della patrolling route in cui $r = v$. In seguito, usando il termine *patrolling route*, supporremo incluso anche il caso del best placement.

Definizione 18 *Dato un vertice v di partenza, una* Covering Route *r è un'insieme di vertici tali che, chiamato $A_r(r(i)) = \sum_{h=0}^{i-1} \omega^*_{r(h),r(h+1)}$ il tempo necessario a visitare il nodo $r(i)$ partendo dal nodo $r(0) = v$, utilizzando per ogni coppia di nodi consecutivi $r(h), r(h+1)$ il loro* shortest path *$\omega^*_{r(h),r(h+1)}$, si abbia:*

$$A_r(t) \leq d(t) \quad \forall t \in r$$

ovvero, tutti i target presenti nella covering route (che indicheremo con l'insieme $T(r)$) vengono visitati prima che scada la loro deadline.

Definizione 19 *Dato un vertice v e un segnale s, un* Covering Set *Q è un sottoinsieme di $T(s)$ tale che esista una covering route r tale che $T(r) = Q$.*

E' evidente che, per ogni target attaccato, il difensore sceglierà, se esiste, il covering set che protegga tutto $T(s)$. Se questo non esiste, sceglierà comunque dei covering set $Q \subset T(s)$. Ricordiamoci inoltre che il vertice v è protetto di default poichè presidiato dal difensore, e lo stesso per i target pattugliati dal difensore, nel caso scegliesse come rotta di default una covering route.

In compenso, per ogni covering set esistono diverse covering route... Scegliere dunque come azioni del giocatore i covering set invece delle covering route corrispondenti potrebbe eliminare alcune soluzioni. Introduciamo quindi i concetti di dominanza per le covering route.

Definizione 20 (Dominanza intra-set) *Date 2 diverse covering route r, r' per (v, s) tale che $T(r) = T(r')$, se $c(r) < c(r')$, allora r domina r'.*

Definizione 21 (Dominanza inter-set) *Date 2 diverse covering route r, r' per (v, s), se $T(r') \subset T(r)$, allora r domina r'.*

E' evidente che, dato un covset, il giocatore sceglierà ovviamente la rotta non dominata che lo copre. Dunque, possiamo tranquillamente sostituire i covset alle covering route come azioni del giocatore. Definiamo ora le utilità dei giocatori, in modo tale da ottenere un gioco a somma 1, come visto in Sezione 2.3. Le utilità dell'attaccante saranno:

- $\pi(t)$, se l'attaccante ha scelto il target t, è scattato l'allarme s, e il difensore ha scelto un covering set Q tale che $t \notin Q$. Nel caso in cui non sia scattato l'allarme, invece, l'attaccante prende suddetta utilità se $t \neq v$ o, nel caso il difensore abbia una covering route r di default, se $t \notin T(r)$.

- 0, altrimenti (poichè viene catturato).

Le utilità per il difensore sono (1−utilità dell'attaccante), ovviamente. In Tabella 3.1 abbiamo un esempio, con le utilità viste dal punto dell'attaccante. La funzione $I(t_i, Q_j)$ vale 0 se $t_i \in Q_j$ (e quindi il ladro viene catturato), $\pi(t_i)$ altrimenti. Con R indichiamo la possibile patrolling route.

	t_1	...	t_n
Q_1	$I(t_1, Q_1)$	...	$I(t_n, Q_1)$
...	...	...	...
Q_m	$I(t_1, Q_m)$	...	$I(t_n, Q_m)$
R	$I(t_1, R)$	...	$I(t_n, R)$

Tabella 3.1: Bimatrice di un PSG, data la rotta R.

3.1.5 SRG-v: complessità

Abbiamo descritto nella sezione precedente come risolvere un SRG-v, senza però tener conto dell'effort computazionale necessario. Presentiamo quindi qui, rimandando a [6] per la dimostrazione, un importante teorema.

Teorema 1 *Sia data un'istanza di un problema SRG-v. Il problema decisionale "Esiste una strategia per il difensore che gli garantisca di guadagnare almeno un'utilità pari a k?" è* $\mathcal{NP}$-HARD.

La difficoltà del problema ci porterà quindi a valutare la possibile scalabilità degli algoritmi di risoluzione del gioco.

3.2 Obiettivi della ricerca

Ora che abbiamo ben formulato il problema, vogliamo capire come cambiano i PSG con l'aggiunta dei falsi negativi. In particolare:

Domanda 1 *Qual è la complessità di risoluzione di un PG? Ovvero, qual è la complessità del problema decisionale "Dato un grafo G=(V,E), con un insieme di target T, di cui sono note le deadline, esiste un covering cycle sull'intero insieme T?"*

In questo caso, la risposta ci viene direttamente da [16], dove si dimostra che il problema è PSPACE-completo.

Definizione 22 *Un problema si dice PSPACE-completo se è risolvibile utilizzando una quantità di memoria polinomiale rispetto all'istanza del problema (ovvero è PSPACE) e se ogni altro problema PSPACE è riducibile in tempo polinomiale ad esso.*

Domanda 2 *Qual è l'algoritmo migliore per risolvere il problema? Qual è la sua scalabilità?*

Domanda 3 *Nel caso un algoritmo esatto non esistesse o fosse impraticabile, qual è la migliore euristica? Quanto si avvicina al valore ottimo?*

Domanda 4 *Quando al difensore conviene fermarsi in un best placement, e quando conviene invece giocare una patrolling route? Come dipende dalla percentuale di falsi negativi α?*

Risponderemo a tutte queste domande nella Sezione 5.

Capitolo 4

Algoritmi

"Al: Sistemare il treppiedi modulare...
John: Fatto!
Al: Avvitare il bullone A con...
John: Fatto!
Al: Assicurate...
John: Fatto!
Al: Inser...
John: Fatto!
Al: Agga...
John: Fatto!
Al: Ins...
John: Fatto! E voilà!"

Aldo, Giovanni e Giacomo, Tre uomini e una gamba

In questa sezione descriveremo tutti gli algoritmi usati per risolvere i vari problemi e giochi, sia in modo esatto che in modo euristico.

4.1 SRG: Ricerca dei Covering Set

Come abbiamo visto in Sezione 3.1.4, possiamo risolvere un SRG scritto in forma strategica, utilizzando come azioni dell'attaccante la scelta del target, e come azioni del difensore i possibili covering set. Ma dato un vertice di partenza v e un segnale s, come possiamo trovare un covering set? E qual è la sua complessità computazionale?

Teorema 2 *Data un'istanza di un problema SRG-v, e un insieme di target T, il problema decisionale "T è un covering set per una qualche covering route?" è $\mathcal{NP}$-completo. (Vedasi prova in [6])*

Ci aspettiamo dunque che un algoritmo che risolve il problema in modo esatto abbia complessità non polinomiale.

4.1.1 Algoritmo esatto: dynamic programming

Questo algoritmo, presentato in [6], utilizza un approccio di *dynamic programming* per trovare iterativamente i covset. Chiamiamo $C^k_{v,t}$ un insieme di covering set $Q^k_{v,t}$, ovvero sequenze di target $\{v, \dots, t\}$ (ove v è il primo nodo visitato, e t l'ultimo), con cardinalità k, tale che esista una covering route r che li copra tutti. Partiamo quindi da $k = 1$, e procediamo iterativamente. Per $k = 1$, ovvero il caso base, la risoluzione è banale, in quanto ogni target ammette una covering route su sè stesso (in pratica, si trasforma in un best placement). Ma come eseguiamo il passo iterativo?

Supponiamo quindi di avere risolto il problema per $Q^{k-1}_{v,t}$. Per ottenere un covering set di cardinalità k, basta cercare un insieme Q^+ di target, tale che $\forall t' \in Q^+$:

- $t' \notin Q^{k-1}_{v,t}$, ovvero non è già stato visitato;
- se aggiunto alla più corta covering route r che copre $Q^{k-1}_{v,t}$, viene raggiunto entro $d(t')$.

In pratica, la nuova covering route r' che copre $Q^k_{v,t}$ sarà semplicemente la vecchia covering route r, a cui viene aggiunto come ultimo nodo t'. Similmente, chiamato $c(Q^k_{v,t})$ il costo temporale della più corta covering route che copre il covering set, e $\omega^*_{t,t'}$ lo shortest path tra t e t', si ha che

$$c(Q^k_{v,t'}) = c(Q^{k-1}_{v,t}) + \omega^*_{t,t'}$$

Possiamo vedere lo pseudocodice dell'algoritmo in Algoritmo 1. Si noti che la funzione SEARCH(Q,C) cerca il covering set Q nell'insieme C per vedere se l'insieme trovato è già presente e, se sì, si tiene quello con costo minore.

- Input
  - I, l'istanza del problema (comprendente il grafo $G = (V, E)$, l'insieme dei target T, l'insieme dei segnali S e le proprietà $d(t)$ e $p(s|t)$ per ogni $t \in T$ e $s \in S$);
  - v, il vertice in cui si trova il difensore;
  - s, il segnale d'allarme a cui rispondere.

- Output
  - $C^k_{v,t}, \forall k \in \{2, \dots, |T(s)|\}$, ovvero tutti i covering set che partono da v e rispondono al segnale s.

Algorithm 1 Risoluzione esatta CovSet

```
1:  procedure ExactCovSet(I,v,s)
2:      ∀t ∈ T(s), k ∈ {2,...,|T(s)|}, C^1_{v,t} = {t}, C^k_{v,t} = ∅
3:      ∀t ∈ T(s), c({t}) = ω*_{v,t}, c(∅) = ∞
4:      for all k ∈ {2,...,|T(s)|} do
5:          for all t ∈ T(s) do
6:              for all Q^{k-1}_{v,t} ∈ C^{k-1}_{v,t} do
7:                  Q+ ← {t' ∈ T(s) \ Q^{k-1}_{v,t} | c(Q^{k-1}_{v,t}) + ω*_{t,t'} ≤ d(t')}
8:                  for all t' ∈ Q+ do
9:                      Q^k_{v,t'} ← Q^{k-1}_{v,t} ∪ {t'}
10:                     U ← Search(Q^k_{v,t'}, C^k_{v,t'})
11:                     if c(U) > c(Q^{k-1}_{v,t}) + ω*_{t,t'} then
12:                         C^k_{v,t'} ← C^k_{v,t'} ∪ {Q^k_{v,t'}}
13:                         c(Q^k_{v,t'}) ← c(Q^{k-1}_{v,t}) + ω*_{t,t'}
14:                     end if
15:                 end for
16:             end for
17:         end for
18:     end for
19:     return ∀k ∈ {1,...,|T(s)|}, C^k_{v,t}
20: end procedure
```

Qual è la complessità dell'algoritmo? Nel caso peggiore, l'algoritmo gira fino a |T(s)|, e dunque il numero di operazioni diventa

$$\sum_{i=1}^{|T(s)|} \binom{|T(s)|}{i-1} (|T(s)| - 1)i$$

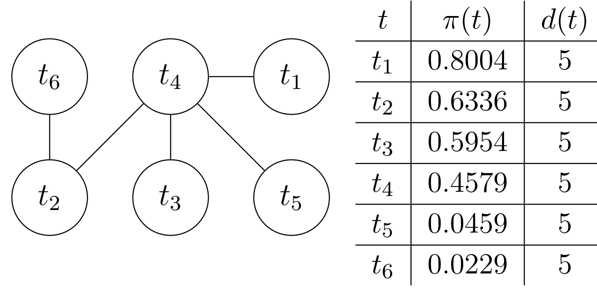
il cui limite superiore è

$$O(|T(s)|^2 2^{|T(s)|})$$

che è non polinomiale, come effettivamente ci aspettavamo.

Esempio 1 *Per chiarire meglio il funzionamento dell'algoritmo, applichiamolo a un grafo reale. Il grafo in Figura 4.1 è stato generato casualmente,*

ed è una delle istanze che verranno poi analizzate nella Sezione 5. Il grafo ha 6 target, le cui deadline e i cui valori sono in Figura 4.1. Tutti gli archi sono bidirezionali, e hanno peso unitario. Tutti i target sono coperti da un solo segnale d'allarme.



t	$\pi(t)$	$d(t)$
t_1	0.8004	5
t_2	0.6336	5
t_3	0.5954	5
t_4	0.4579	5
t_5	0.0459	5
t_6	0.0229	5

Figura 4.1: Esempio di grafo da pattugliare

Supponiamo che il difensore sia nel target t_4. Quindi inizializziamo tutte le collezioni C^1 con i vari target, in quanto tutti i target sono raggiungibili da t_4 entro le loro deadline. Dunque abbiamo

$$C^1_{4,1} = \{t_1\},\ C^1_{4,2} = \{t_2\},\ C^1_{4,3} = \{t_3\},\ C^1_{4,5} = \{t_5\},\ C^1_{4,6} = \{t_6\}$$

Poichè eseguendo tutto il codice avremmo troppi insiemi da controllare, concentriamoci solo sull'insieme $C^1_{4,6}$, di costo 2[11]*. In questo caso, Q^+ è composto dai target t_1 $(2+3 \leq 5)$, t_2 $(2+1 \leq 5)$, t_3 $(2+3 \leq 5)$, t_5 $(2+3 \leq 5)$. I nuovi insiemi ottenuti sono così:*

$$Q^2_{4,1} = \{t_6, t_1\},\ Q^2_{4,2} = \{t_6, t_2\},\ Q^2_{4,3} = \{t_6, t_3\},\ Q^2_{4,5} = \{t_6, t_5\}$$

Per tutti gli insiemi tranne $Q^2_{4,2}$, $Q^+ = \emptyset$, in quanto il loro costo è già pari a 5, ovvero alla deadline dei target. Per $Q^2_{4,2}$, abbiamo invece $Q^+ = \{t_1, t_3, t_5\}$. I nuovi insiemi sono quindi

$$Q^3_{4,1} = \{t_6, t_2, t_1\},\ Q^3_{4,3} = \{t_6, t_2, t_3\},\ Q^3_{4,5} = \{t_6, t_2, t_5\}$$

tutti di costo pari a 5. Poichè $Q^+ = \emptyset$, l'algoritmo termina, in quanto non esistono più altri insiemi da controllare.

[11] Banalmente, la covering route è $\{t_4, t_2, t_6\}$.

4.1.1.1 Eliminazione dei covset dominati

Si noti che l'algoritmo appena descritto fornisce tutti i possibili covset, dato un vertice v e un segnale a cui rispondere s. Il numero di covset è molto grande, e inoltre forniamo anche covset le cui covering route sono palesemente dominate da quelle di altri covset.

Supponiamo quindi di voler marcare alcuni covset come dominati, e di rimuoverli alla fine dell'esecuzione dell'algoritmo[12]. A ogni iterazione, posso trovare un covset Q^k che domina tutti gli insiemi Q^{k-1} per la dominanza inter-set. Al massimo, ho $|Q|$ insiemi dominati, da cercare però in ogni collezione $C_{v,t}^{k-1}$, per ogni target t, il cui caso pessimo è $|T(s)|$. La ricerca dell'insieme nella collezione può essere effettuata in $O(|T(s)|)$, utilizzando un albero binario. Facendo i dovuti calcoli, si ottiene un onere computazionale aggiuntivo di $O(|T(s)|^3)$, riducendo però di molto il numero di covset generati e dunque la complessità computazionale della risoluzione del SRG-v. La complessità dell'algoritmo con eliminazione dei covset dominati diventa dunque

$$O(|T(s)|^5 2^{|T(s)|})$$

4.1.2 Euristica : monotonic covering route

Come abbiamo visto nella sezione precedente, trovare i covering set in modo esatto è molto oneroso, e dunque l'algoritmo verrà utilizzato solo per piccole istanze. Utilizziamo dunque un algoritmo approssimato, sempre presentato in [6], che si basa sulle *monotonic covering route.*

Definizione 23 *Un covering route r si dice monotona rispetto a un ordinamento O quando, dato un ordinamento totale sui target $O : t_1 < \cdots < t_n$, r rispetta tale ordinamento, ovvero, $\forall t, t' \in r$, t può precedere t' in r solo se $t < t'$.*

Si può dimostrare (vedi [6]) che trovare una *monotonic covering route,* dato un particolare ordinamento, è un problema risolvibile in tempo polinomiale. Noi proveremo 4 possibili ordinamenti[13]:

- rispetto agli shortest path $\omega_{v,t}^*$. E' più probabile che i target t più vicini a v siano più facili da coprire entro la loro deadline $d(t)$.

[12] Non è possibile rimuoverli prima, in quanto si possono perdere dei covset non dominati.
[13] I possibili pareggi vengono risolti in modo randomico.

- rispetto alle deadline $d(t)$. E' più probabile che i target con deadline strette vadano visitati il prima possibile.
- rispetto all'*excess time*, ovvero il tempo rimanente per coprire un target t dopo aver percorso lo shortest path da v a t. Più è stretta la differenza $d(t) - \omega^*_{v,t}$, prima andrà visitato il target.
- randomico, con $nPerm$ rilanci.

In Algoritmo 2, presentiamo lo pseudocodice dell'algoritmo che calcola tutte le *monotonic covering route*, dato un ordinamento totale sui target. Sfruttiamo ancora una volta l'approccio del dynamic programming. Sia $R(k, l)$ una matrice che contiene, in ogni cella, una rotta che copre l target, partendo dal target t_k,[14] e sia $L(k, l)$ la matrice che contiene per la rotta contenuta in $R(k, t)$ la sua *lateness* massima, ovvero la massima differenza su tutti i target in r tra il tempo di arrivo in t seguendo r e la sua deadline $d(t)$ (in pratica, quanto arrivo "in ritardo" per proteggere il target t). Ovviamente, cercando delle covering route non dominate, memorizzeremo in $R(k, l)$ la rotta con lateness minima.

Ma come costruiamo $R(k, l)$? Utilizziamo il dynamic programming. Il caso base, per $l = 1$, è banalmente la coppia $R(k, 1) = (v, t_k)$. Per costruire invece $R(k, l)$, aggiungiamo alla rotta $R(k, 1)$ tutte le rotte $R(k', l - 1)$, con $k' > k$ per mantenere l'ordinamento O. L'algoritmo restituisce quindi tutte le rotte presenti in R, da cui ricavare i covering set e dunque le azioni del difensore. La complessità dell'algoritmo è $O(|T(s)|^3)$, escludendo il calcolo dei shortest path (che possono comunque essere precalcolati in $O(|T|^3)$).

- Input
  - I, l'istanza del problema (comprendente il grafo $G = (V, E)$, l'insieme dei target T, l'insieme dei segnali S e le proprietà $d(t)$ e $p(s|t)$ per ogni $t \in T$ e $s \in S$);
  - v, il vertice in cui si trova il difensore;
  - s, il segnale d'allarme a cui rispondere;
  - O, l'ordinamento totale su tutti i target (O_R sarà il suo ordine inverso).
- Output
  - R, matrice contenente tutte le *monotonic covering route* trovate.

Algorithm 2 MonotonicLongestRoute

1: **procedure** MONOTONICLONGESTROUTE(I,v,s,O)
2: $\forall k, k' \in O, R(k,k') = \emptyset, L(k,k') = +\infty, C_R(k) = \emptyset, C_L(k) = +\infty$
3: **for all** $\forall k \in O_R$ **do**
4: **for all** $\forall l \in \{1, 2, \ldots, |T(s)|\}$ **do**
5: **if** $l = 1$ **then**
6: $R(k,l) = \langle v, t_k \rangle$
7: $L(k,l) = w^*_{v,t_k} - d(t_k)$
8: **else**
9: **for all** k' that follows k in O **do**
10: $C_R(k') = \langle R(k,1), R(k', l-1) \rangle$
11: $C_L(k') = \max\{L(k,1), w^*_{v,t_k} + w^*_{t_k,k'} - w^*_{v,k'} + L(k', l-1)\}$
12: **end for**
13: $j = \arg\min_j \{C_L(j)\}$
14: **if** $C_L(j) \leq 0$ **then**
15: $R(k,l) \leftarrow C_R(j)$
16: $L(k,l) \leftarrow C_L(j)$
17: **end if**
18: **end if**
19: **end for**
20: **end for**
21: **return** R
22: **end procedure**

L'algoritmo finale utilizzato raccoglierà poi tutte le azioni del difensore ottenute tramite i 4 ordinamenti sopra descritti, il cui ultimo dipende fortemente dal numero di rilanci $nPerm$ scelto. Riportiamo lo pseudocodice per completezza in Algoritmo 3.

- Input
  - I, l'istanza del problema (comprendente il grafo $G = (V, E)$, l'insieme dei target T, l'insieme dei segnali S e le proprietà $d(t)$ e $p(s|t)$ per ogni $t \in T$ e $s \in S$);
  - v, il vertice in cui si trova il difensore;
  - s, il segnale d'allarme a cui rispondere;
  - $nPerm$, il numero di rilanci per l'ordinamento randomico. Se pari a 0, suddetto ordinamento non viene considerato.
- Output
  - actions, le azioni del difensore (ovvero i possibili covset trovati).

Algorithm 3 Risoluzione approssimata CovSet

```
 1: procedure PermCovSet(I,v,s,nPerm)
 2:     actions ← ∅
 3:     for all O ∈ Orders do                    ▷ i 3 ordinamenti non randomici
 4:         newActions ← MonotonicLongestRoute(I, v, s, O)
 5:         actions ← actions ∪ newActions
 6:     end for
 7:     for i←1 to nPerm do                       ▷ ordinamento randomico
 8:         O ← random order
 9:         newActions ← MonotonicLongestRoute(I, v, s, O)
10:         actions ← actions ∪ newActions
11:     end for
12:     return actions
13: end procedure
```

Esempio 2 *Applichiamo l'algoritmo nuovamente al grafo in Figura 4.1. Supponiamo sempre che il difensore parta dal nodo* t_4*, e supponiamo di avere un ordine casuale* $O = \{t_4, t_6, t_5, t_2, t_1, t_3\}$*, e dunque dal suo reverso* $O_R = \{t_3, t_1, t_2, t_5, t_6, t_4\}$*. Inizializziamo l'algoritmo, avendo così:*

[14] Per semplicità chiameremo t_k il k-simo target secondo l'ordinamento O.

$$R(1,1) = \{t_4, t_1\},\ R(2,1) = \{t_4, t_2\},\ R(3,1) = \{t_4, t_3\},\ R(5,1) = \{t_4, t_5\},\ R(6,1) = \{t_4, t_6\}$$

$$L(1,1) = -4,\ L(2,1) = -4,\ L(3,1) = -4,\ L(5,1) =,\ L(6,1) = -3$$

Passiamo alla seconda iterazione dell'algoritmo. Anche qui, per semplicità, analizziamo solo $R(5,2)$. I possibili candidati da aggiungere sono tutti quelli del passo precedente, escluso $R(5,1)$ e $R(6,1)$, in quanto t_6 deve essere visitato prima di t_5. Calcoliamo le lateness, chiamandole impropriamente col nome del target k' aggiunto:

$$L(1) = -2,\ L(2) = -2,\ L(3) = -2$$

In questo caso, le lateness sono tutte identiche, e il pareggio viene risolto in modo randomico. Supponiamo, ad esempio, che venga aggiunto il target t_2. Per calcolare $R(5,3)$ ci servono $R(2,2)$, $R(1,2)$ e $R(3,1)$, che possiamo facilmente verificare essere $\{t_2, t_1\}$ (oppure indifferentemente $\{t_2, t_3\}$),$\{t_1, t_3\}$ e $\{t_3\}$. Poichè all'iterazione l si sceglie l'insieme di cardinalità $l-1$, scartiamo l'ultimo insieme. Gli altri due hanno ancora la stessa lateness, dunque scegliamo randomicamente $R(2,2)$, ottenendo così $R(5,3) = \{t_4, t_5, t_2, t_1\}$. L'ultima iterazione è su $R(5,4)$, la cui unica possibilità di aggiunta è l'insieme $R(2,3) = \{t_2, t_1, t_3\}$. Così facendo, però, sforiamo la deadline, violando i costraint di covset. Abbiamo dunque ottenuto il covset massimo ottenibile partendo dal vertice t_4 e utilizzando come primo target t_5. Il procedimento è molto più veloce rispetto all'algoritmo esatto, ma fa purtroppo perdere delle soluzioni come, ad esempio, il covset $\{t_4, t_6, t_2, t_5\}$, in quanto il target t_5 è costretto ad essere visitato prima del target t_2, così come il target t_6.

4.2 SRG-v: Risoluzione del gioco

Abbiamo visto come si può rappresentare un SRG-v nella Sezione 3.1.4, e come la sua soluzione sia il valore *minmax* del gioco. Ma come si trova algoritmicamente suddetta soluzione?

Adottiamo una tecnica che rientra nella branca della ricerca operativa: la programmazione lineare (LP) [11]. Un problema in programmazione lineare è composto da una funzione obiettivo $f(x)$, del quale cerchiamo il valore massimo o minimo, e da alcuni vincoli sulle variabili, generalmente definiti nella forma $Ax \leq b$, dove A è la matrice dei coefficienti, b il vettore dei parametri e x il vettore delle variabili. Nel nostro caso abbiamo:

- variabili: x, y, v, ovvero le strategie del primo e del secondo giocatore, e il valore minmax del gioco;
- coefficienti: P, ovvero la matrice $m \times n$ dei payoff del gioco (dove p_{ij} è l'utilità del gioco se il primo giocatore gioca la riga i, e il secondo la colonna j).

Il problema viene dunque formulato come:

$$
\begin{array}{llll}
\text{maximize} & v & & \\
\text{subject to} & xP \geq v \cdot 1_n, & & \text{(primo giocatore)} \\
 & Py \leq v \cdot 1_m, & & \text{(secondo giocatore)} \\
 & & & \\
 & \sum_{x_i \in X} x_i = 1, & i = 1, ..., m & \text{(probabilità)} \\
 & \sum_{y_j \in Y} y_j = 1, & j = 1, ..., n & \\
 & x_i \geq 0, & i = 1, ..., m & \text{(non negatività)} \\
 & y_j \geq 0, & j = 1, ..., n &
\end{array}
$$

Cosa significa tutto questo? Il primo giocatore vuole ottenere il massimo payoff possibile, ma per effetto del secondo giocatore, prenderà il massimo dei minimi. La sua strategia, dunque, deve fornire un payoff maggiore o uguale (se esiste, esattamente uguale) al valore minmax del gioco. Lo stesso, invertito, per il secondo giocatore, che prenderà il minimo dei massimi. Le altre quattro equazioni sono semplicemente imposte per rendere x e y delle probabilità, dunque la somma delle loro componenti, che devono essere non negative, deve essere 1. Possiamo scegliere se massimizzare o minimizzare indifferentemente v. Il tutto dipende per quale giocatore lo stiamo risolvendo. In questo caso, scegliendo il primo, stiamo massimizzando.

Non entreremo nei dettagli di come risolvere un problema di LP, in quanto non necessario per la nostra ricerca.[15] Se necessario, useremo la dicitura LPSOLVER negli pseudocodici per indicare la risoluzione di un problema di LP. Lo pseudocodice per la risoluzione di un SRG-v è infatti, banalmente:

- Input
  - I, l'istanza del problema (comprendente il grafo $G = (V, E)$, l'insieme dei target T, l'insieme dei segnali S e le proprietà $d(t)$ e $p(s|t)$ per ogni $t \in T$ e $s \in S$);

[15] Lasciamo comunque un riferimento al metodo del simplesso, uno dei metodi più noti ed usati [10].

  - α, la percentuale di falso negativo che affligge i sensori;
  - v, il vertice in cui si trova il difensore.

- Output

  - gameValue, l'utilità del nodo per il difensore.

Algorithm 4 SRG-v

Require: $0 \leq \alpha \leq 1$

1: **procedure** SRGV(I,α,v)
2: $covsets \leftarrow$ COVSET(I, α, v)
3: $gameValue \leftarrow$ LPSOLVER($covsets, T, P$)
4: **return** $gameValue$
5: **end procedure**

4.2.1 SRG-cycle: Risoluzione del gioco

Abbiamo appena visto come risolvere un SRG-v. In questo modo, possiamo dare un valore ai vertici e utilizzare la *backward induction* per risolvere il gioco a ritroso. Ma per risolvere il gioco, oltre a dare un valore ai vertici, dobbiamo trovare un modo per dare un valore anche alle possibili rotte di pattuglia del difensore. Possiamo definire l'utilità di un covering cycle del difensore come:

- uguale all'utilità attesa del difensore, ottenuta risolvendo il corrispettivo SRG-v, se il covering cycle è, ovviamente, composto da un solo nodo (best placement).

- uguale al minimo delle utilità attese del difensore, ottenute risolvendo i vari SRG-v per ogni vertice appartenente al covering cycle. Perchè proprio il minimo? Semplice, perchè l'attaccante aspetterà di entrare in azione quando vedrà il difensore nella posizione a lui più congeniale, e dunque il valore di tutto il ciclo si ridurrà al valore del suo "nodo più debole".

Lo pseudocodice dell'algoritmo (piuttosto banale, ma riportato per completezza) è visibile in Algoritmo 5, dove:

- Input

  - I, l'istanza del problema (comprendente il grafo $G = (V, E)$, l'insieme dei target T, l'insieme dei segnali S e le proprietà $d(t)$ e $p(s|t)$ per ogni $t \in T$ e $s \in S$);
  - α, la percentuale di falso negativo che affligge i sensori;
  - cycle, il covering cycle da valutare.

- Output

  - cycleValue, l'utilità del covering cycle rispetto al difensore.

Algorithm 5 SRG-cycle

Require: $0 \leq \alpha \leq 1$

1: **procedure** SRGcycle(I,α,cycle)
2:     $cycleValue \leftarrow 1$
3:     **for all** $v \in cycle$ **do**
4:         $currentValue \leftarrow \text{SRGV}(I, \alpha, v)$
5:         **if** $currentValue < cycleValue$ **then**
6:             $cycleValue \leftarrow currentValue$
7:         **end if**
8:     **end for**
9:     **return** $cycleValue$
10: **end procedure**

4.3 PG: Enumerazione possibili covering cycles

Il modo esatto per risolvere un PG è enumerare tutti i possibili sottoinsiemi di nodi, verificare se esiste un covering cycle e, se esiste, valutarne la sua utilità. Ma se calcolare l'utilità di un ciclo è molto semplice (al più $|c|$ risoluzioni di SRG-v, dove c è il nostro covering cycle), eseguire suddetto calcolo per tutti i possibili sottoinsiemi di nodi del grafo è molto oneroso (è in pratica il *power set* 2^n). Dovremo quindi cercare di sfruttare le informazioni insite nella topologia del grafo o nelle strategie dei giocatori per cercare di ridurre lo spazio di ricerca.

4.3.1 Algoritmo esatto: enumerazione completa

Come già detto, in questo caso enumeriamo completamente tutti i sottoinsiemi di nodi del grafo, cerchiamo un covering cycle, lo valutiamo, e procediamo fino ad esaurimento dei sottoinsiemi. Come già anticipato, la complessità

temporale dell'algoritmo sarà $O(2^n)$ a causa del *power set*. In Algoritmo 6 mostriamo lo pseudocodice dell'algoritmo, avente come dati:

- Input
  - I, l'istanza del problema (comprendente il grafo $G = (V, E)$, l'insieme dei target T, l'insieme dei segnali S e le proprietà $d(t)$ e $p(s|t)$ per ogni $t \in T$ e $s \in S$);
  - α, la percentuale di falso negativo che affligge i sensori.
- Output
  - bestCycle, il miglior covering cycle trovato.

Algorithm 6 Enumerazione completa

Require: $0 \leq \alpha \leq 1$
1: **procedure** BestcycleExact(I,α)
2:     $bestCycle \leftarrow \emptyset$
3:     $bestCycleValue \leftarrow 0$
4:     **for all** $\tau \subseteq T$ **do**
5:         $cycle \leftarrow$ SearchCycle(I, τ)
6:         **if** $cycle \neq \emptyset$ **then**
7:             $currentValue \leftarrow$ SRGcycle$(I, \alpha, cycle)$
8:             **if** $currentValue > bestCycleValue$ **then**
9:                 $bestCycle \leftarrow cycle$
10:                 $bestCycleValue \leftarrow currentValue$
11:             **end if**
12:         **end if**
13:     **end for**
14:     **return** $bestCycle$
15: **end procedure**

Esempio 3 *Utilizziamo sempre il grafo in Figura 4.1. L'algoritmo esatto enumera tutti i possibili sottoinsiemi dei 6 target, quindi le* $2^6 - 1 = 63$ *possibili combinazioni sono, ordinate per cardinalità:*

- $\{t_1\}, \{t_2\}, \{t_3\}, \{t_4\}, \{t_5\}, \{t_6\}$
- $\{t_1, t_2\}, \{t_1, t_3\}, \{t_1, t_4\}, \{t_1, t_5\}, \{t_1, t_6\}, \{t_2, t_3\}, \{t_2, t_4\}, \{t_2, t_5\}, \{t_2, t_6\}, \{t_3, t_4\}, \{t_3, t_5\}, \{t_3, t_6\}, \{t_4, t_5\}, \{t_4, t_6\}, \{t_5, t_6\}$

- $\{t_4,t_5,t_6\},\{t_3,t_5,t_6\},\{t_3,t_4,t_6\},\{t_3,t_4,t_5\},\{t_2,t_5,t_6\},\{t_2,t_4,t_6\},\{t_2,t_4,t_5\},$ $\{t_2,t_3,t_6\},\{t_2,t_3,t_5\},\{t_2,t_3,t_4\},\{t_1,t_5,t_6\},\{t_1,t_4,t_6\},\{t_1,t_4,t_5\},\{t_1,t_3,t_6\},$ $\{t_1,t_3,t_5\},\{t_1,t_3,t_4\},\{t_1,t_2,t_6\},\{t_1,t_2,t_5\},\{t_1,t_2,t_4\},\{t_1,t_2,t_3\}$
- $\{t_3,t_4,t_5,t_6\},\{t_2,t_4,t_5,t_6\},\{t_2,t_3,t_5,t_6\},\{t_2,t_3,t_4,t_6\},\{t_2,t_3,t_4,t_5\},\{t_1,t_4,t_5,t_6\},$ $\{t_1,t_3,t_5,t_6\},\{t_1,t_3,t_4,t_6\},\{t_1,t_3,t_4,t_5\},\{t_1,t_2,t_5,t_6\},\{t_1,t_2,t_4,t_6\},\{t_1,t_2,t_4,t_5\},$ $\{t_1,t_2,t_3,t_6\},\{t_1,t_2,t_3,t_5\},\{t_1,t_2,t_3,t_4\}$
- $\{t_1,t_2,t_3,t_4,t_5\},\{t_1,t_2,t_3,t_4,t_6\},\{t_1,t_2,t_3,t_5,t_6\},\{t_1,t_2,t_4,t_5,t_6\},\{t_1,t_3,t_4,t_5,t_6\},$ $\{t_2,t_3,t_4,t_5,t_6\}$
- $\{t_1,t_2,t_3,t_4,t_5,t_6\}$

4.3.1.1 Riduzione complessità temporale: la tabu list

Come abbiamo visto, l'algoritmo esatto è molto pesante in termini di complessità computazionale. Ciononostante, è possibile ridurre lo spazio di ricerca tramite una *tabu list* molto semplice: se un sottoinsieme S di nodi non ammette un covering cycle, allora nemmeno il sottoinsieme $S \cup v$, con $v \notin S$, ammette un covering cycle, dunque non deve essere nemmeno considerato.

Supponendo quindi di creare tutti i possibili sottoinsiemi di nodi in ordine di cardinalità crescente (prima quelli da un nodo, poi quelli da 2 e così via), possiamo immettere nella tabu list tutti i sottoinsiemi che non ammettono covering cycle, e controllare a ogni iterazione che il nostro sottoinsieme S non contenga uno dei sottoinsiemi presenti nella tabu list. Si noti però che questo aumenta la complessità spaziale dell'algoritmo, in quanto la tabu list può crescere, al peggio, esponenzialmente, e comporta comunque un certo peso computazionale durante lo scorrimento della suddetta alla ricerca di un possibile sottoinsieme del nostro insieme S.[16] Per semplicità, abbiamo scelto di non utilizzare suddetta ottimizzazione, che potrà essere implementata in ricerche future.

4.3.2 Euristica statica: diametro massimo del sottografo

Poichè l'algoritmo esatto di enumerazione dei possibili covering cycle ha una complessità esponenziale, cerchiamo di ridurre lo spazio degli stati di ricerca utilizzando delle euristiche.

Definizione 24 *Un'euristica si dice* statica *se dipende solo dall'istanza del problema, e se non cambia durante la risoluzione del problema. In caso contrario, prende il nome di euristica* dinamica.

[16] Suddetto peso si può evitare utilizzando una tabella hash, che però deve introdurre altre complessità nella scelta della funzione di hash e nel mantenimento della tabella.

Iniziamo quindi con un'euristica statica, che cerca di sfruttare la topologia del grafo. L'idea è che i nodi più vicini tra di loro siano più facilmente copribili da un covering cycle, in quanto si "sprechi" meno tempo a raggiungerli. Preso un insieme S di nodi (ovvero un sottografo di I), utilizziamo la nozione di diametro di un grafo per avere un'idea della vicinanza dei nodi appartenenti ad S.

Definizione 25 *Dato un grafo $G = (V, E)$, il* diametro *di G è il più grande shortest path tra tutte le coppie di nodi $(u, v) \in V^2$.*

Impostiamo quindi un parametro k e cerchiamo tutti i possibili sottoinsiemi di nodi S tali che il loro diametro sia minore o uguale a k. Anche qui utilizziamo un approccio in dynamic programming. Chiamiamo C_i la collezione che contiene tutti gli insiemi S_i di cardinalità i. Il caso base è semplice, in quanto ogni singolo target ha, ovviamente, diametro inferiore a k, visto che è un singolo nodo. Dunque $C_1 = T$, banalmente. Il passo iterativo è altrettanto semplice. Prendiamo ogni sottografo $S_i \in C_i$ e ne calcoliamo l'insieme $J = T \setminus S_i$. Per ogni $j \in J$, otteniamo $S_{i+1} = S_i \cup \{j\}$. Controlliamo poi se il diametro di S_{i+1} è minore o uguale a k. Se lo è, lo aggiungiamo alla collezione C_{i+1}, altrimenti lo scartiamo. L'algoritmo termina quando non ci sono più sottografi da analizzare.

Come possiamo subito notare, impostando un k troppo alto porterà a scartare ben pochi sottografi, riportandoci così nel caso dell'algoritmo esatto, e dunque una complessità esponenziale. Vedremo poi nella Sezione 5.4 come ovviare al problema utilizzando una versione anytime dell'algoritmo. Lo pseudocodice della versione descritta è visibile in Algoritmo 7.

- Input
  - I, l'istanza del problema (comprendente il grafo $G = (V, E)$, l'insieme dei target T, l'insieme dei segnali S e le proprietà $d(t)$ e $p(s|t)$ per ogni $t \in T$ e $s \in S$);
  - α, la percentuale di falso negativo che affligge i sensori;
  - k, la lunghezza massima del diametro dei sottografi.
- Output
  - bestCycle, il miglior covering cycle trovato.

Algorithm 7 Euristica statica: distanza

```
Require: k ≥ 1, 0 ≤ α ≤ 1
 1: procedure HeurDist(I,α,k)
 2:     C_1 ← T
 3:     for i ← 2 to |T| do
 4:         C_i ← ∅
 5:     end for
 6:     for i ← 1 to |T| − 1 do
 7:         for all S_i ∈ C_i do
 8:             J ← T \ S_i
 9:             for all j ∈ J do
10:                 S_{i+1} ← S_i ∪ {j}
11:                 if Diameter(S_i) ≤ k then        ▷ Valuta il diametro di S_i
12:                     C_{i+1} ← C_{i+1} ∪ S_{i+1}
13:                 end if
14:             end for
15:         end for
16:     end for
17:     C ← C_1 ∪ ··· ∪ C_{|T|}
18:     bestCycle ← ∅
19:     bestCycleValue ← 0
20:     for all c ∈ C do
21:         cycle ← SearchCycle(I, c)
22:         if cycle ≠ ∅ then
23:             currentValue ← SRGcycle(I, α, cycle)
24:             if currentValue > bestCycleValue then
25:                 bestCycle ← cycle
26:                 bestCycleValue ← currentValue
27:             end if
28:         end if
29:     end for
30:     return bestCycle
31: end procedure
```

Esempio 4 *Applichiamo l'algoritmo al grafo in Figura 4.1, supponendo di utilizzare come distanza massima $k = 1$. Inizializziamo l'algoritmo, ottenendo così*

$$C_1 = T = \{\{t_1\}, \{t_2\}, \{t_3\}, \{t_4\}, \{t_5\}, \{t_6\}\}$$

Costruiamo ora C_2 iterativamente. Di tutte le possibili coppie che includono t_1, solo $\{t_1, t_4\}$ ha un diametro minore o uguale di 1. Lo stesso per t_3 e t_5, che forniscono dunque le coppie $\{t_3, t_4\}$ e $\{t_5, t_4\}$. t_2 ci fornisce invece le coppie $\{t_2, t_6\}$ e $\{t_2, t_4\}$. t_6 e t_4 non possono fornire che coppie già presenti in C_2, che diventa quindi

$$C_2 = \{\{t_1, t_4\}, \{t_3, t_4\}, \{t_5, t_4\}, \{t_2, t_4\}, \{t_2, t_6\}, \}$$

Proviamo ora a costruire C_3, ma senza successo. Non vi sono infatti più target da aggiungere senza aumentare il diametro del grafo sopra il k imposto, e dunque l'algoritmo termina. Se avessimo imposto $k = 2$ avremmo ottenuto tutti i possibili sottoinsiemi, esclusi quelli che includono contemporaneamente t_6 e uno dei target t_1,t_3 e t_5, mentre per $k = 3$ avremmo ottenuto tutti i possibili sottoinsiemi.

4.3.3 Euristica statica: ordinamento dei target per valore

Presentiamo qui un'altra possibile euristica statica, stavolta cercando di sfruttare l'ipotesi di egoismo dei giocatori. Poichè entrambi i giocatori cercheranno di prendere la maggiore utilità possibile, è molto più probabile che, tra 2 target t, t', con valori $\pi(t) > \pi(t')$, l'attaccante cercherà di colpire il target con valore maggiore, ovvero t. Il difensore, per converso, cercherà invece di proteggere suddetti target. Proviamo quindi a ordinare i target in base al loro valore $\pi(t)$. Proviamo quindi a proteggere il target più importante. Se esiste la rotta, proviamo a proteggere il più importante, e il secondo più importante, e così via.

Ma in questo modo eliminiamo veramente troppe soluzioni, riducendo il gioco alla ricerca di una rotta su un solo ordinamento di target. Introduciamo quindi un parametro k, che denota la *tolleranza* con la quale scegliamo i target da inserire nei possibili covering cycle. Cosa significa questo parametro? Semplice, che invece di prendere solo il target a valore maggiore, prenderemo il target a valore maggiore e i k target seguenti, provando tutte le combinazioni possibili la cui lunghezza dipende dall'iterazione dell'euristi-

ca. All'iterazione i avremo infatti tutte le combinazioni dei primi $i+k$ target in i-tuple[17]. Ogni iterazione proveremo quindi

$$\binom{i+k}{i}$$

sottoinsiemi S, ottenendo così un numero di operazioni pari a

$$\sum_{i\in[1,|T|]}\binom{i+k}{i}$$

Si noti che, ponendo $k=0$ l'euristica diventa addirittura lineare ($O(|T|)$), ma riduce l'insieme delle possibili rotte a una sola rotta non dominata, mentre ponendo $k=|T|$ ritorniamo all'algoritmo esatto. Vediamo lo pseudocodice dell'algoritmo in Algoritmo 8, ove con COMBNOVERI(O,n,i) intendiamo tutte le combinazioni dei primi n target rispetto all'ordinamento O nelle i-ple.

- Input
  - I, l'istanza del problema (comprendente il grafo $G=(V,E)$, l'insieme dei target T, l'insieme dei segnali S e le proprietà $d(t)$ e $p(s|t)$ per ogni $t\in T$ e $s\in S$);
  - α, la percentuale di falso negativo che affligge i sensori;
  - k, la tolleranza nel prendere i target durante le combinazioni.
- Output
  - bestCycle, il miglior covering cycle trovato.

Esempio 5 *Prendiamo i nodi del grafo in Figura 4.1. I target sono già ordinati alfabeticamente per valore (t_1 ha il maggior valore, e t_6 il minore). Mostriamo solamente due casi: il primo con tolleranza pari a 0, il secondo con tolleranza pari a 1. Nel primo caso, otterremo tutte le combinazioni del tipo n target da posizionare in n posizioni, dunque avremo*

$$C=\{\{t_1\},\{t_1,t_2\},\{t_1,t_2,t_3\},\{t_1,t_2,t_3,t_4\},\{t_1,t_2,t_3,t_4,t_5\},\{t_1,t_2,t_3,t_4,t_5,t_6\}\}$$

Nel secondo caso, invece, avremo tutte le combinazioni del tipo $n+1$ in n posizioni, dunque avremo

[17] Ovviamente, se $i+k>|T|$, verrà scelto $|T|$ come numeratore.

Algorithm 8 Euristica statica: valore

Require: $k \geq 0, 0 \leq \alpha \leq 1$

```
 1: procedure HeurValue(I,α,k)
 2:     C ← ∅
 3:     bestCycle ← ∅
 4:     O ← targets t_i ordered by their value π(t_i)
 5:     for i ← 1 to |T| do
 6:         n ← Min(i + k, |T|)
 7:         C_i ← CombNoverI(O, n, i)
 8:         C ← C ∪ C_i
 9:     end for
10:     for all c ∈ C do
11:         cycle ← SearchCycle(I, c)
12:         if cycle ≠ ∅ then
13:             currentValue ← SRGcycle(I, α, cycle)
14:             if currentValue > bestCycleValue then
15:                 bestCycle ← cycle
16:                 bestCycleValue ← currentValue
17:             end if
18:         end if
19:     end for
20:     return bestCycle
21: end procedure
```

$C = \{\{t_1\}, \{t_1, t_2\}, \{t_1, t_3\}, \{t_2, t_3\}, \{t_1, t_2, t_3\}, \{t_1, t_2, t_4\}, \{t_1, t_3, t_4\}, \{t_2, t_3, t_4\}, \{t_1, t_2, t_3, t_4\}, \{t_1, t_2, t_3, t_5\}, \{t_1, t_2, t_4, t_5\}, \{t_1, t_3, t_4, t_5\}, \{t_2, t_3, t_4, t_5\}, \{t_1, t_2, t_3, t_4, t_5\}, \{t_1, t_2, t_3, t_4, t_6\}, \{t_1, t_2, t_3, t_5, t_6\}, \{t_1, t_2, t_4, t_5, t_6\}, \{t_1, t_3, t_4, t_5, t_6\}, \{t_2, t_3, t_4, t_5, t_6\}, \{t_1, t_2, t_3, t_4, t_5, t_6\}\}$

4.3.4 Euristica dinamica: strategia dell'attaccante

Dopo aver mostrato due euristiche statiche, passiamo ora ad un'euristica dinamica. Riprendendo un po' il concetto dell'euristica statica che ordina i target per il loro valore, cerchiamo anche qui di dare un certo ordinamento ai target, ma basandoci stavolta sulle strategie dei giocatori.

L'idea è la seguente. Supponiamo che inizialmente il difensore non si muova, e dunque che la sua mossa migliore sia di rimanere nel best placement v. A meno di casi particolari, che comunque non sono presenti per $\alpha > 0$, l'attaccante sceglierà sempre di attaccare uno o più target, che sono quelli che massimizzano la sua utilità.[18] Sembrerebbe dunque logico che il difensore cerchi di proteggere i target prediletti dall'attaccante, e dunque li aggiungiamo all'insieme dei possibili target su cui cercare la nostra covering cycle. Se la troviamo, il tutto si ripete nuovamente, fino ad esaurimento dei possibili sottografi esplorabili oppure ad esaurimento delle scelte dell'attaccante, che potrebbe attaccare dei nodi già valutati oppure che non può attaccare più nulla, in quanto tutti i nodi sono protetti.

Purtroppo, per come è descritta attualmente, l'euristica non è completa. Ovvero, può terminare senza garanzie di aver esplorato tutto lo spazio degli stati. Per ottenere ciò, introduciamo anche qui un parametro k che indica la tolleranza nella scelta dei prossimi target da controllare, ovvero, prenderemo tutti i target che danno utilità massima all'attaccante e i primi k target seguenti, ordinati per utilità dell'attaccante decrescente. Inoltre, poichè vogliamo che vengano visitati per primi i sottoinsiemi S composti solo da nodi con utilità dell'attaccante massima, assegniamo ad ogni sottoinsieme una priorità $h(S)$, che parte da $h(\emptyset) = 0$ e viene costruita iterativamente nel seguente modo[19]:

[18] Utilità che possiamo facilmente ricavare dalla risoluzione del LP presentato in Sezione 4.2.

[19] Esistono molti modi di aggiornare iterativamente l'euristica. Noi descriveremo ed useremo la somma, ma altre possibili alternative sono ad esempio il minimo o il massimo dei valori.

$$h(S \cup \{t\}) = \begin{cases} h(S), & \text{se il target } t \text{ rende massima l'utilità dell'attaccante} \\ h(S)+1, & \text{altrimenti} \end{cases}$$

dove, ovviamente, più basso è il valore e più alta è la priorità, e dove eventuali pareggi vengono risolti tramite una logica FIFO (ovvero, il primo che viene trovato, viene messo in cima alla coda). In Algoritmo 9 mostriamo lo pseudocodice dell'algoritmo.

- Input
  - I, l'istanza del problema (comprendente il grafo $G = (V, E)$, l'insieme dei target T, l'insieme dei segnali S e le proprietà $d(t)$ e $p(s|t)$ per ogni $t \in T$ e $s \in S$);
  - α, la percentuale di falso negativo che affligge i sensori;
  - k, la tolleranza nel prendere i target durante le combinazioni.
- Output
  - bestCycle, il miglior covering cycle trovato.
- Funzioni helper [20]
  - AttackerBestChoices(S,α), restituisce i target colpiti dall'attaccante quando il difensore protegge in target in S, avendo un falso negativo pari ad α;
  - AttackerOtherChoices(S,α,k), restituisce i primi k target, ordinandoli secondo l'utilità dell'attaccante, non colpiti dall'attaccante quando il difensore protegge i target in S, avendo un falso negativo pari ad α;
  - GetValue(targets,others), fornisce i valori ai target in $\{targets \cup others\}$. In particolare, dà valore 0 ai targets in targets e 1 a quelli in others;
  - GetHeuristic(S,t,h,v), calcola la funzione $h(S \cup t)$;
  - OrderedInsert(S,C), inserisce l'insieme S in C ordinandolo secondo il valore della funzione $h(S)$;
  - Pop(C), restituisce il primo elemento di C;
  - MaxValue(C), restituisce la miglior rotta presente in C.

Algorithm 9 Euristica dinamica

Require: $k \geq 0, 0 \leq \alpha \leq 1$

1: **procedure** HeurDyn(I,α,k)
2: $bp \leftarrow$ BestPlacement(I, α) ▷ Restituisce il BestPlacement
3: $targets \leftarrow$ AttackerBestChoices(bp, α)
4: $others \leftarrow$ AttackerOtherChoices(bp, α, k)
5: $visited \leftarrow bp$
6: $v(targets \cup others) \leftarrow$ GetValues$(targets, others)$
7: $v(bp) \leftarrow 0$
8: $h(targets \cup others \cup bp) \leftarrow v(targets \cup others \cup bp)$
9: $toVisit \leftarrow \emptyset$
10: **for all** $(i, j) \in targets \cup others \cup bp$ **do**
11: $h(i, j) \leftarrow$ GetHeuristic(i, j, h, v)
12: $toVisit \leftarrow$ OrderedInsert$((i, j), toVisit)$
13: **end for**
14: **while** $toVisit \neq \emptyset$ **do**
15: $S \leftarrow$ Pop$(toVisit)$
16: $cycle \leftarrow$ SearchCycle(S)
17: **if** $cycle \neq \emptyset$ **and** $S \notin visited$ **then**
18: $visited \leftarrow visited \cup S$
19: $targets \leftarrow$ AttackerBestChoices(S, α)
20: $others \leftarrow$ AttackerOtherChoices(S, α, k)
21: **for all** $t \in targets \cup others$ **do**
22: $h(S \cup t) \leftarrow$ GetHeuristic$(S, t, h, v, heurMode)$
23: OrderedInsert$(S \cup t, toVisit)$
24: **end for**
25: **end if**
26: **end while**
27: $bestCycle \leftarrow$ MaxValue$(visited)$
28: **return** $bestCycle$
29: **end procedure**

Esempio 6 *Applichiamo l'algoritmo al grafo in Figura 4.1, risolvendo il gioco per* α *pari a* 0.5 *e con* $k = 0$.[21] *Iniziamo calcolando il best placement, come visto in Sezione 4.2. Scopriamo così che il best placement, ovvero la miglior rotta di un solo nodo, è il target* t_1. *Quai sono dunque i migliori target da attaccare per l'attaccante? Quelli per cui l'utilità del difensore è minima. Le utilità sono, in ordine alfanumerico:*

$$[1.0000, 0.6832, 0.7005, 0.7711, 0.9542, 0.9885]$$

Il target sotto attacco, dunque, è il target t_2. *Cerchiamo dunque un covering cycle che copra i target* t_1 *e* t_2 *(vedremo poi in Sezione 4.4 come fare). Troviamo quindi il covering cycle* $\{t_1, t_4, t_2, t_4, t_1\}$, *e ripetiamo il procedimento. Le utilità del difensore sono, stavolta*

$$[1.0000, 0.8909, 0.7023, 1.0000, 0.9620, 0.9846]$$

dunque, il nodo sotto attacco è il nodo t_3. *Purtroppo non esiste un covering cycle che copra tutti e tre i target, dunque l'algoritmo termina.*

4.4 PG: Ricerca covering cycle

Dopo aver visto diversi modi per enumerare i possibili sottoinsiemi S di nodi su cui cercare un covering cycle, affrontiamo finalmente il problema principale: dato un insieme di nodi S, esiste un covering cycle che copra tutti i target in S?

Prima di tutto, definiamo matematicamente le proprietà di un covering cycle. Chiamiamo

- σ, la sequenza di nodi che compone il nostro covering cycle, dove $\sigma(i)$ è l'i-esimo elemento del ciclo e $s = |\sigma|$;
- $w(i, j)$, il peso dell'arco (i, j);
- $O_i(j)$ la posizione in σ della j-esima occorrenza del target i;
- o_i il numero di occorrenze del vertice i in σ;

[20] Il codice è molto corposo e molte funzioni dipendono molto dall'implementazione e dalle strutture dati usate. Tralasceremo quindi il loro pseudocodice e le descriveremo qui a parole.

[21] Poichè la tolleranza k è posta a 0, non dovremo calcolare i valori della funzione h, che saranno sempre pari a 0.

- a' è la matrice di adiacenza.

Abbiamo dunque che σ è un covering cycle per un insieme di target T se sono valide le seguenti condizioni:

$$\sigma(1) = \sigma(s) \tag{4.1}$$

$$o_i \geq 1 \qquad \forall i \in T \tag{4.2}$$

$$a'\left(\sigma(j-1), \sigma(j)\right) = 1 \qquad \forall j \in \{2, 3, \dots, s\} \tag{4.3}$$

$$\sum_{j=O_i(k)}^{O_i(k+1)-1} w\left(\sigma(j), \sigma(j+1)\right) \leq d(i) \qquad \forall i \in T, \forall k \in \{1, 2, \dots, o_i - 1\} \tag{4.4}$$

$$\sum_{j=1}^{O_i(1)-1} w\left(\sigma(j), \sigma(j+1)\right) + \sum_{j=O_i(o_i)}^{s-1} w\left(\sigma(j), \sigma(j+1)\right) \leq d(i) \qquad \forall i \in T \tag{4.5}$$

ovvero

- (4.1), il primo e l'ultimo elemento coincidono, essendo un ciclo;
- (4.2), tutti i target in T sono presenti almeno una volta nel ciclo;
- (4.3), i nodi in σ sono raggiungibili rispettando la sequenza;
- (4.4), se esistono più occorrenze di uno stesso vertice i, allora il tempo intercorso tra ogni coppia di occorrenze è minore o uguale alla sua deadline;
- (4.5), come (4.4), ma considerando l'ultima occorrenza e la prima, in quanto il ciclo deve essere valido anche durante la sua riesecuzione.

Purtroppo, come dimostrato in [7], il problema "dato un insieme di target T, esiste un covering cycle che copra tutti i target in T?" è $\mathcal{NP}$-*COMPLETO*. L'algoritmo di risoluzione esatto, dunque, avrà complessità sicuramente non polinomiale.

4.4.1 Algoritmo esatto: backtracking con forward checking

Questo algoritmo, presentato sempre in [7], si basa banalmente sul backtracking. Il funzionamento è semplice: imponiamo una lunghezza massima k del covering cycle che stiamo cercando, e poi proviamo a inserire nei nodi nella sequenza σ. Il primo nodo è indifferente, poichè è un ciclo. Se durante l'iterazione i tutti i costraint sono validi, allora restituiamo la rotta, altrimenti proviamo o ad aggiungere altri nodi adiacenti al target in $\sigma(i)$, oppure

eseguiamo backtracking se abbiamo già un ciclo completo di tutti i target, ma che non soddisfano i constraint sulle deadline.

La complessità dell'algoritmo è, banalmente, $O(|T|^k)$, in quanto ad ogni iterazione ho da inserire, alla peggio, $|T|-1$ nodi. A prima vista, sembra una complessità polinomiale, ma poichè $k \geq |T|+1$, altrimenti non potremmo mai avere un covering cycle, possiamo scrivere $k = \varepsilon|T|$, con $\varepsilon > 1$. Abbiamo dunque una complessità esponenziale $O(|T|^{\varepsilon|T|})$.

Come possiamo fare per ridurre questa complessità? Cerchiamo di ridurre il numero di nodi che è possibile inserire nella sequenza all'iterazione i utilizzando il forward checking. L'idea è questa: verifichiamo anticipatamente i vincoli (4.4) e (4.5), ed eliminiamo i nodi che non li soddisfano già all'iterazione i. Poichè non abbiamo i veri pesi degli archi nella sequenza, in quanto non conosciamo la sequenza a priori, utilizziamo lo *shortest path*, che indicheremo con $\bar{w}$. Se il vincolo non è soddisfatto già con lo shortest path fra i nodi, sicuramente non sarà soddisfatto nemmeno dalla sequenza con i tempi reali. Poichè lo shortest path è una sottostima del tempo reale, non scarteremo alcuna soluzione possibile.

Di seguito, lo pseudocodice dell'algoritmo.

- Input
  - I, l'istanza del problema (comprendente il grafo $G = (V, E)$, l'insieme dei target T, l'insieme dei segnali S e le proprietà $d(t)$ e $p(s|t)$ per ogni $t \in T$ e $s \in S$);
  - T, l'insieme di target su cui cercare il covering cycle;
  - k, lunghezza massima del covering cycle;
  - iteration, iterazione corrente;
  - σ, sequenza attuale di nodi.
- Output
  - σ, il covering cycle trovato, oppure $\emptyset$ se non è stato trovato;
  - $F_{iteration}$, i possibili nodi da aggiungere alla sequenza attuale di nodi.

Algorithm 10 Ricerca di un covering cycle

Require: $k \geq |T| + 1$

```
1: procedure SearchCycle(I,T,k)
2:     σ(1) ← Random(T)                              ▷ Il target iniziale è casuale
3:     σ ← SearchCycleRecursive(I, T, k, 2)
4:     return σ
5: end procedure
```

Algorithm 11 Ricerca di un covering cycle: chiamata ricorsiva

Require: $k \geq |T| + 1$

```
1: procedure SearchCycleRecursive(I,T,k,iteration)
2:     if iteration > k then
3:         return ∅
4:     end if
5:     if σ(1) = σ(iteration − 1) and costraint (4.2) verificato then
6:         if costraint (4.5) verificato then
7:             return σ
8:         else
9:             return ∅
10:        end if
11:    else
12:        F_iteration ← ForwardChecking(I, T, σ, iteration)
13:        for all t ∈ F_iteration do
14:            σ(iteration) ← t
15:            σ′ ← SearchCycleRecursive(I, T, k, iteration + 1)
16:            if σ′ ≠ ∅ then
17:                return σ′
18:            end if
19:        end for
20:    end if
21: end procedure
```

Algorithm 12 Ricerca di un covering cycle: forward checking

Require: $k \geq |T| + 1$

1: **procedure** ForwardChecking(I,T,σ,iteration)
2: $F_{iteration} \leftarrow \emptyset$
3: $s \leftarrow iteration - 1$
4: **for all** $i \in T$ adiacenti a $\sigma(s)$ **do**
5: **if** valgono le seguenti condizioni:
6: $\Big(o_i = 0 \wedge \sum_{l=1}^{s-1} w(\sigma(l), \sigma(l+1)) + w(\sigma(s), i) + \overline{w}(i, \sigma(1)) \leq d(i)$ or
7: $o_i > 0 \wedge \sum_{l=O_i(o_i)}^{s-1} w(\sigma(l), \sigma(l+1)) + w(\sigma(s), i) \leq d(i)\Big)$ and,
8: for all $k \neq i$,
9: $\Big(o_k = 0 \wedge \sum_{l=1}^{s-1} w(\sigma(l), \sigma(l+1)) + w(\sigma(s), i) + \overline{w}(i, k) + \overline{w}(k, \sigma(1)) \leq d(k)$ or
10: $o_k > 0 \wedge \sum_{l=O_k(o_k)}^{s-1} w(\sigma(l), \sigma(l+1)) + w(\sigma(s), i) + \overline{w}(i, k) \leq d(k)\Big)$
11: **then**
12: $F_{iteration} \leftarrow F_{iteration} \cup i$
13: **end if**
14: **end for**
15: **return** $F_{iteration}$
16: **end procedure**

Esempio 7 *Applichiamo l'algoritmo al grafo in Figura 4.1. Supponiamo di voler cercare un covering cycle che copra, ad esempio, i target* t_1,t_2 *e* t_4. *Scegliamo randomicamente il nodo iniziale del nostro ciclo, prendendo per esempio* t_2, *e applichiamo il forward checking. I target adiacenti sono* t_4 *e* t_6, *e rispettano tutti i vincoli del forward checking, dunque scegliamo randomicamente, tra i due,* t_4. *La nostra sequenza diventa quindi*

$$\sigma = \{t_2, t_4\}$$

Procediamo iterativamente. Da t_4 *abbiamo come possibili target* t_1,t_2,t_3,t_4, *ma* t_2 *non rispetta i vincoli del forward checking, e viene quindi scartato. Scegliamo tra i rimanenti* t_1 *in modo randomico. Abbiamo così*

$$\sigma = \{t_2, t_4, t_1\}$$

Da t_1 *possiamo andare solo in* t_4, *che rispetta i vincoli del forward checking. La nostra sequenza diventa quindi*

$$\sigma = \{t_2, t_4, t_1, t_4\}$$

Da t_4, abbiamo di nuovo t_1,t_2,t_3,t_4 come possibili candidati, ma stavolta è t_1 a non rispettare i vincoli di forward checking, e viene dunque scartato. Scegliamo randomicamente t_2. La nostra sequenza è ora

$$\sigma = \{t_2, t_4, t_1, t_4, t_2\}$$

e, poichè è un ciclo, e rispetta tutti i vincoli sulle deadline, è il covering cycle che stavamo cercando.

Capitolo 5

Analisi sperimentale

> *"Giovanni: Allora, esercitazione, eh? Facciamo conto che c'è stata una chiamata nel cuore della notte, perché in un appartamento di via Giovanni da Procida si sentono urla, schiamazzi, musica ad alto volume, rumori molesti ed anche odori molesti.(...) Sugar, come agiamo?"*
> *Giacomo: Intanto, un buon poliziotto dorme già vestito, con tuta mimetica, l'elmetto, gli anfibi e qualche bomba a mano sotto il cuscino.*
> *Giovanni: Bravo!*
> *Giacomo: Al primo squillo di telefono so già dove recarmi.*
> *Giovanni: Bene!*
> *Giacomo: Esco in strada e comincio a sparare all'impazzata.*
> *Giovanni: Per quale motivo?*
> *Giacomo: Perché se sono sveglio io, non vedo perché gli altri devono dormire!*
>
> Aldo, Giovanni e Giacomo, Tel chi el telun

In questa sezione descriveremo tutti gli esperimenti eseguiti per rispondere alle domande presentate precedentemente nella Sezione 3.2.

5.1 Setting sperimentale

Prima di descrivere gli esperimenti in sè, bisogna definire il tipo di istanze utilizzate, il tipo di parametri scelti e bisogna fornire tutti i dettagli implementativi.

Iniziamo a descrivere le istanze utilizzate. Per essere significative, le istanze non devono essere nè troppo semplici, altrimenti non potremmo notare il comportamento dell'algoritmo sotto sforzo, nè troppo difficili, in quanto ci troveremmo a sprecare tempo per poi non ottenere alcun tipo di risultato.

Vogliamo inoltre che le nostre istanze siano quanto meno verosimili. Evitiamo dunque di generare in modo perfettamente casuale i grafi, ma poniamo dei vincoli, ovvero

- il grafo $G = (V, E)$ è composto solo da target, in quanto semplifica l'analisi;
- ogni arco ha costo unitario;
- tutte le deadline sono fissate a $n - 1$, dove $n = |V|$;
- l'incertezza α non supera il 50%, altrimenti i sensori sarebbero fin troppo irrealistici;
- tutti i target sono coperti da un solo segnale d'allarme;
- la densità del grafo, ovvero il rapporto $\frac{2|E|}{n(n-1)}$, sarà del 25% o del 75%, in quanto realisticamente è difficile trovare grafi meno o più connessi di queste due densità.

I parametri rimanenti vengono generati casualmente da una distribuzione uniforme. In seguito, ci riferiremo alle istanze a seconda della loro dimensione, ovvero il numero di target, della densità e dell'incertezza.

Parlando invece dell'implementazione, tutti gli algoritmi sono stati sviluppati in MATLAB [3], compreso il codice per generare le istanze e i grafici dei risultati. Gli esperimenti sono stati eseguiti su una macchina LINUX 8-core, con CPU da 2.33GHz e 16 GB di RAM.

5.2 Esperimento: ricerca covering cycles

Abbiamo già dimostrato nella Sezione 3.2 che il problema di decidere se esiste o meno un covering cycle su un insieme di target sia PSPACE-completo, e abbiamo visto nella Sezione 4.4 che l'algoritmo di ricerca dei covering cycle, nel caso pessimo, abbia complessità $O(|T|^{k|T|})$. Ciò che vogliamo analizzare è, invece, come scala la qualità dell'algoritmo rispetto al parametro k utilizzato. Poichè non è possibile utilizzare un approccio qualitativo sulle rotte trovate, in quanto la campagna sperimentale diventerebbe troppo lunga e complessa da gestire[22], valuteremo la qualità dell'algoritmo in modo quantitativo: quante rotte riesce a trovare l'algoritmo, dato un certo k, entro una

[22]Dovremmo infatti confrontare, oltre alle rotte, anche il loro valore, e dunque dovremmo anche risolvere effettivamente il gioco.

determinata quantità di tempo?

Organizziamo la nostra campagna sperimentale usando:[23]

- 50 grafi, per ottenere risultati statisticamente rilevanti;
- la dimensione del grafo presa dall'insieme $\{6, 8, 10, 12, 20, 30, 40, 50\}$, per verificare la scalabilità dell'algoritmo;
- la densità del grafo pari a 0.25 e a 0.75, come anticipato in Sezione 5.1;
- un'ora come limite temporale massimo di esecuzione per ogni grafo e per ogni k, per questioni pratiche;
- k appartenente all'insieme $\{0.5,1,2\}$. In pratica, utilizziamo come bound la metà, il doppio o esattamente la dimensione del grafo stesso.

5.2.1 Risultati: ricerca covering cycles

Nelle Figure 5.1 e 5.2 mostriamo i risultati, mediati sui 50 grafi utilizzati, delle analisi svolte sulle istanze, rispettivamente, con densità pari a 0.25 e a 0.75.

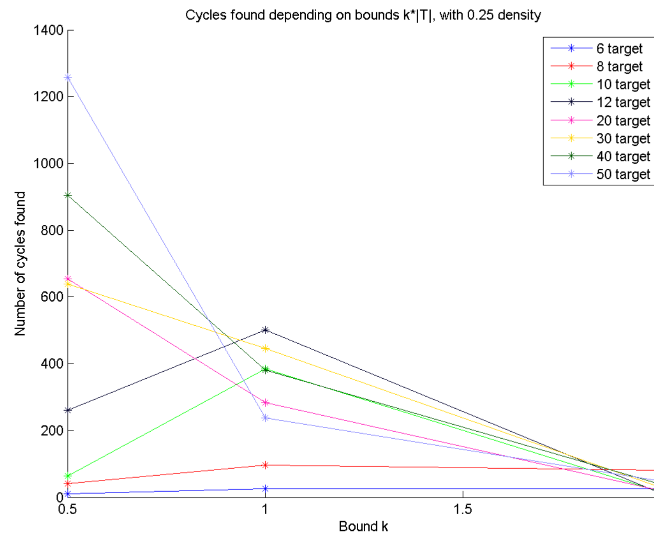


Figura 5.1: Covering cycles trovati su istanze di densità pari a 0.25 *a seconda del bound k scelto.*

[23] α non viene utilizzato in questa analisi, in quanto la ricerca delle rotte è indipendente dall'incertezza dei sensori.

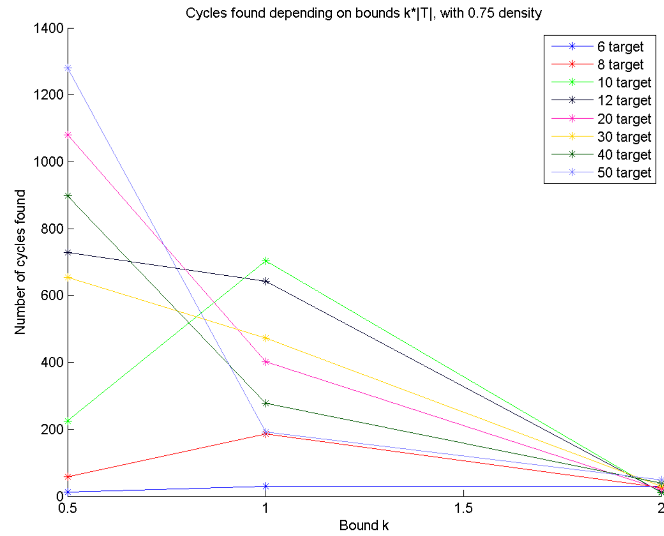


Figura 5.2: Covering cycles trovati su istanze di densità pari a 0.75 *a seconda del bound k scelto.*

Possiamo notare come, in realtà, la densità del grafo non influenzi la relazione tra il bound utilizzato e le rotte trovate, quanto piuttosto influenzi solamente il numero di rotte possibili, e dunque di rotte trovate. Un grafo molto connesso, infatti, ha potenzialmente più rotte di un grafo meno connesso, in quanto esistono più modi per connettere i nodi. Sebbene questo influenzi anche il processo di backtracking, in quanto si hanno più rami nell'albero, il parametro chiave rimane comunque il bound scelto.

La scelta del bound è un processo molto delicato, e dipende infatti fortemente dalla dimensione del grafo. Possiamo infatti notare che per grafi relativamente piccoli, come ad esempio quelli di dimensione 6, le rotte crescano al crescere di k, mentre si ha l'effetto opposto per grafi grandi, come ad esempio quelli dai 20 target in su, nei quali le rotte crescono al decrescere di k. Come caso intermedio, invece, abbiamo i grafi da 8, 10 e 12 target, che hanno un massimo per $k = 1$.

Tutto questo è facilmente spiegabile. Se il grafo è piccolo, l'albero di backtracking nel caso peggiore non è comunque abbastanza lungo da diventare computazionalmente oneroso. Già con 8, 10 e 12 target iniziamo a notare un incremento della complessità computazionale, infatti ponendo $k = 2$ perdiamo delle rotte trovate con $k = 1$. Questo è dato dal fatto che di-

mostrare che una rotta non esiste su un particolare insieme di target richiede l'esplorazione completa dell'albero, e dunque ponendo $k = 2$ perdiamo troppo tempo a dimostrare che le rotta non esiste. Questo effetto è ancora più evidente sui grafi da 20 target in su, che risentono del problema già con $k = 1$.

Una possibile soluzione a tutto questo, che però non verrà qui analizzata, è cercare un buon compromesso tra rotte trovate e tempo utilizzato, ed utilizzare una specie di regressione per ricavare tutti i bound migliori utilizzabili a seconda della dimensione del grafo. Ad esempio, supponiamo che un buon compromesso sia utilizzare $k = 0.5$ per i grafi da 50 target. Ricordando che la complessità dell'algoritmo è $O(n^{kn})$, potremmo ricavare i successivi k come $k = log_n(50^{0.5(50)})$. Questo comunque non è oggetto di questa ricerca, e verrà lasciato come possibile studio futuro.

5.3 Esperimento: risoluzione esatta del gioco

Ci chiedevamo, nella Sezione 3.2, quale fosse il miglior algoritmo per risolvere il gioco, quale fosse la sua scalabilità e come dipendesse la cardinalità del covering cycle migliore a seconda dell'incertezza del grafo, con particolare enfasi su quando convenga giocare un best placement, e quando un ciclo di pattuglia più lungo.

In questa sezione esamineremo e risolveremo il gioco in modo esatto, utilizzando quindi gli algoritmi presentati nelle Sezioni 4.1.1, 4.2 e 4.3.1. Come abbiamo già discusso nelle relative Sezioni 4.1.1 e 4.3.1, enumerare esattamente tutti i possibili sottoinsiemi di target ed enumerare in modo esatto i covset ha complessità esponenziale rispetto al numero di target, dunque avremo sicuramente problemi di scalabilità. Ciò che vogliamo analizzare è a che dimensione del grafo iniziamo ad avere problemi, impostando un limite temporale entro il quale, se non viene completato il preprocessing, la risoluzione del grafo viene interrotta e l'esperimento viene considerato fallito.

Bisogna inoltre valutare come cambiano il valore del gioco e la cardinalità della rotta a seconda dell'incertezza presente nei sensori. Ci aspettiamo in particolare che il valore del gioco degradi, in quanto il difensore non è più così ben assistito dagli allarmi, e dunque aumentano le probabilità dell'attaccante di farla franca, mentre non abbiamo particolari previsioni sulla cardinalità. Una constatazione banale è che, se esistesse il covering cycle massimale, verrebbe sempre giocato quello. Ma poichè i nostri grafi sono costruiti con deadline più strette appositamente per evitare questa casisistica,

non abbiamo alcuna previsione in merito.

Organizziamo dunque la nostra campagna sperimentale usando:

- 50 grafi, per ottenere risultati statisticamente rilevanti;
- la dimensione del grafo presa dall'insieme $\{6, 8, 10, 12\}$, in quanto la scalabilità dell'algoritmo è molto labile, e dunque è inutile provarlo su grafi troppo grandi;
- la densità del grafo pari a 0.25 e a 0.75, come anticipato in Sezione 5.1;
- un'ora come limite temporale massimo di esecuzione per ogni grafo e per ogni α, per questioni pratiche;
- α appartenente all'insieme $\{0,0.1,0.2,0.3,0.4,0.5\}$. Come già detto in 5.1, utilizziamo questi parametri per rendere i sensori realistici, dunque evitiamo incertezze troppo alte.

5.3.1 Risultati: risoluzione esatta del gioco

I risultati delle analisi sono visibili in Tabella 5.1 e nelle Figure 5.3, 5.4, 5.5, 5.6. Poichè i grafici sono corposi, analizziamo un aspetto per volta. Facciamo però notare che alcuni grafici, più precisamente quelli per i grafi da 12 target, sono portati a 0, in quanto il tempo non è bastato per risolverli.

5.3.1.1 Risultati: preprocessing

Come possiamo vedere dalla Tabella 5.1, il cosiddetto preprocessing, ovvero il calcolo esatto dei covset e l'enumerazione completa delle rotte, ha una scalabilità molto ridotta. Se non si hanno problemi con grafi piccoli da 6 e 8 target, l'algoritmo esatto inizia a non terminare entro il tempo limite già con i grafi da 10 target. Se con una densità bassa, ovvero il 25%, i grafi non risolti sono ancora accettabili (circa un sesto dei grafi totali), aumentando la densità fino al 75%, la percentuale di grafi non risolti si avvicina molto al 100%. Aumentando ancora la dimensione, e arrivando a 12 target, l'algoritmo non riesce a terminare il preprocessing nemmeno di un grafo.

Possiamo quindi concludere che l'algoritmo esatto è utilizzabile quindi praticamente solo su grafi molto piccoli, con un numero di target inferiore o uguale ad 8. Ricordiamo però che abbiamo impostato arbitrariamente come tempo limite un'ora. Non è oggetto di questa ricerca, ma è possibile che l'algoritmo, aumentando il tempo limite a 2 o 4 ore, riesca comunque

a risolvere dei grafi da 10 target, rendendolo comunque appetibile per la risoluzione di problemi di media dimensione. Diverso discorso per i grafi da 12 target che, come è visibile dai risultati, non sono assolutamente risolvibili in modo esatto in un tempo ragionevole.[24]

		Densità	
		0.25	0.75
Dimensione	6	0%	0%
	8	0%	0%
	10	16%	92%
	12	100%	100%

Tabella 5.1: Percentuale di grafi il cui preprocessing ha ecceduto il tempo limite.

5.3.1.2 Risultati: valori del gioco

Possiamo vedere gli andamenti del valore del gioco rispetto all'incertezza del segnale nelle Figure 5.3 e 5.4. Possiamo notare che, a parte il cambio in valore assoluto del valore del gioco, dovuto alla generazione casuale dei grafi, l'andamento dei grafici è indipendente dalla dimensione e dalla densità del grafo, ma decresce in modo praticamente lineare all'aumentare di α.

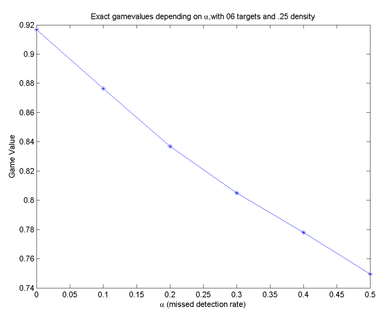
Questo è effettivamente un risultato che ci aspettavamo, in quanto aumentando l'incertezza dei sensori, aumentiamo anche la probabilità che il ladro possa attaccare un target indisturbato. Questo, ovviamente, se il target non è presente nel covering cycle giocato dal difensore che, visti i risultati, ci aspettiamo che cresca all'aumentare dell'incertezza, per poter cercare di proteggere più target possibili.

5.3.1.3 Risultati: cardinalità del covering cycle

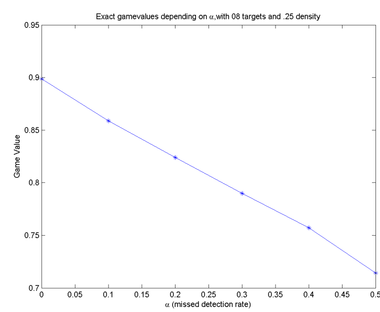
Possiamo vedere come varia la cardinalità[25] del miglior covering cycle rispetto all'incertezza del segnale nelle Figure 5.5, 5.6.

[24]Per scrupolo sono state analizzate due istanze casuali, una da 10 e una da 12 target, senza limiti di tempo. L'istanza da 10 target è stata risolta in circa 4 ore, mentre quella da 12 ci ha impiegato un giorno intero. Questi risultati sono comunque dipendenti dall'unico grafo utilizzato, e dunque hanno solo valore intuitivo, ma non statistico.

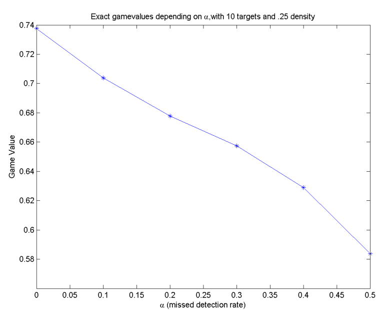
[25]Ricordiamo che per cardinalità indichiamo, impropriamente, il rapporto tra i target coperti dal covering cycle e i target totali del grafo.



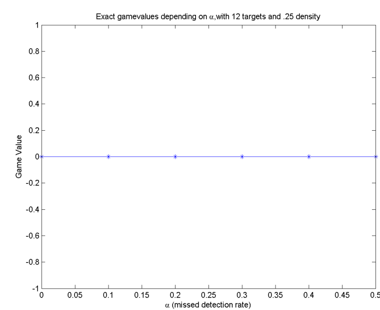
(a) *Grafi a 6 target.*



(b) *Grafi a 8 target.*

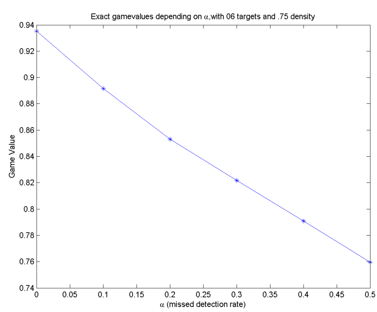


(c) *Grafi a 10 target.*

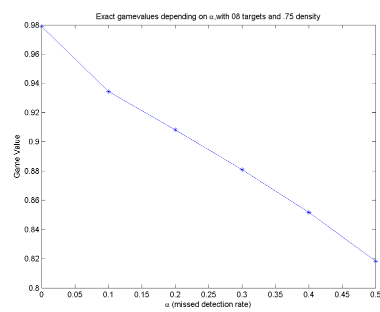


(d) *Grafi a 12 target.*

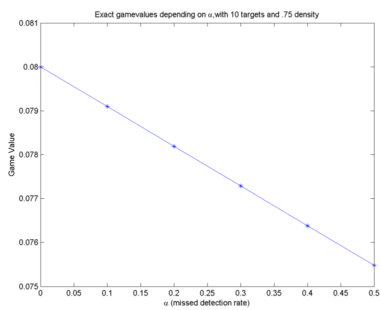
Figura 5.3: Andamento del valore esatto del gioco a seconda dell'incertezza α*, mediato sui grafi con densità del* 25%.



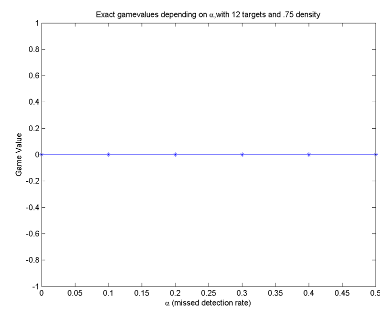
(a) *Grafi a 6 target.*



(b) *Grafi a 8 target.*



(c) *Grafi a 10 target.*



(d) *Grafi a 12 target.*

Figura 5.4: Andamento del valore esatto del gioco a seconda dell'incertezza α, mediato sui grafi con densità del 75%.

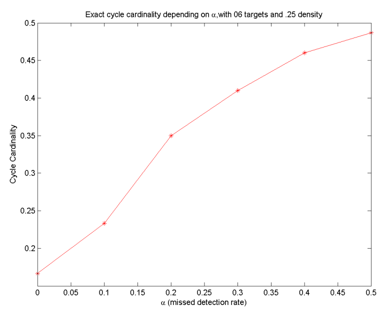
La cardinalità, indipendentemente dalla dimensione del grafo e dalla densità, ha sempre lo stesso andamento. Quando il segnale è perfetto, ovvero $\alpha = 0$, la miglior mossa del difensore è il bestplacement. Infatti, essendo sicuro al 100% che l'allarme suonerà, il difensore si porrà nella miglior posizione di risposta, ovvero nel miglior vertice in cui attendere, e non avrà bisogno di pattugliare. All'aumentare dell'incertezza, però, il difensore inizia a pattugliare per evitare di perdere dei nodi a causa di allarmi non scattati. L'incertezza critica dopo la quale il difensore è spinto a muoversi dipende molto dal grafo, dai valori dei target e dalle deadline, e dunque non ci addentreremo nel discorso. Possiamo inoltre notare che la crescita nella cardinalità non è lineare, anzi, possiamo notare che la curva, dopo una particolare incertezza, tende a non crescere e a stabilizzarsi. Evidentemente a un certo punto non esistono rotte più grandi, oppure ingrandire le possibili rotte di pattuglia porterebbe a tralasciare o indebolire determinati nodi troppo importanti, e dunque ci si focalizza su una rotta di lunghezza fissa, generalmente attorno alla metà dei nodi presenti.

Si può notare che la dimensione e la densità dei grafi non influenzano particolarmente l'andamento della cardinalità, se non per la lunghezza assoluta della rotta o per la pendenza della curva. Inoltre, non avendo risolto il 92% dei grafi da 10 target, e non avendo risolto nemmeno un grafo da 12 target, non abbiamo abbastanza informazioni per eseguire dei confronti statisticamente validi.

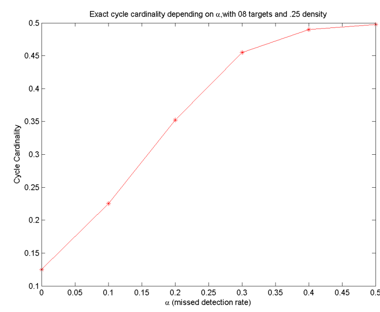
5.4 Esperimento: risoluzione approssimata del gioco

Come abbiamo visto nella Sezione 5.3, risolvere il gioco in maniera completamente esaustiva ed esatta è impraticabile per grafi la cui dimensione supera i 10 target. Dobbiamo quindi cercare di risolvere il gioco in modo più efficiente, senza tuttavia allontanarci troppo dalla soluzione ottima. In questo esperimento testeremo quindi gli algoritmi presentati nelle Sezioni 4.1.2, 4.3.2, 4.3.3 e 4.3.4.

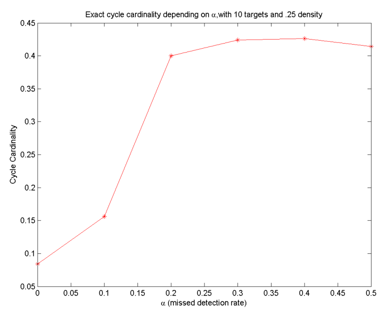
Perchè scegliamo proprio questi algoritmi? E' semplice. Dai risultati visti in Sezione 5.3, notiamo che la risoluzione del grafo fallisce già durante il preprocessing, ovvero durante il calcolo dei covset e dei possibili covering cycle. La risoluzione dell'SRG-v tramite LP non è quindi il collo di botti-



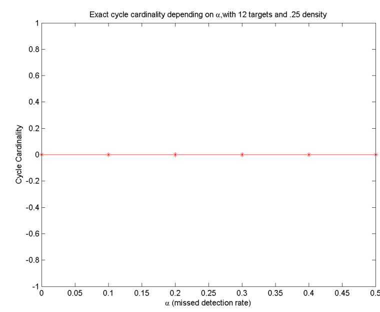
(a) *Grafi a 6 target.*



(b) *Grafi a 8 target.*

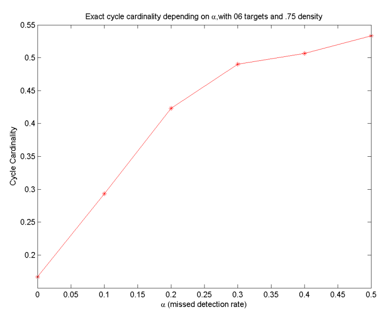


(c) *Grafi a 10 target.*

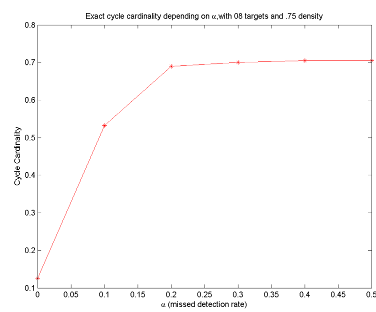


(d) *Grafi a 12 target.*

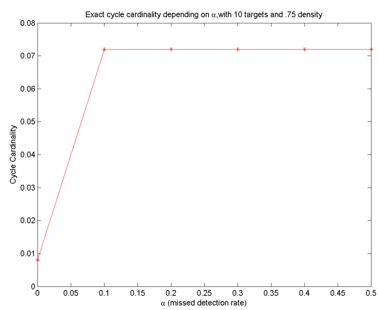
Figura 5.5: Andamento della cardinalità della rotta migliore, calcolata in modo esatto, a seconda dell'incertezza α, mediato sui grafi con densità del 25%*.*



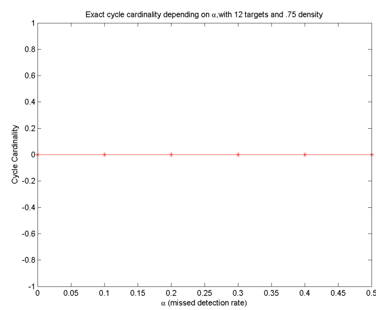
(a) *Grafi a 6 target*



(b) *Grafi a 8 target*



(c) *Grafi a 10 target*



(d) *Grafi a 12 target.*

Figura 5.6: Andamento della cardinalità della rotta migliore, calcolata in modo esatto, a seconda dell'incertezza α, mediato sui grafi con densità del 75%.

glia dell'algoritmo, e manterremo quindi la risoluzione esatta. Al contrario, siamo costretti a provare delle approssimazioni polinomiali per il calcolo dei covset e dei covering cycle.

Useremo l'algoritmo presentato in Sezione 4.1.2 per approssimare il calcolo dei covset, utilizzando un tempo polinomiale nel numero di rilanci. Poichè anche in questo caso vogliamo analizzare la scalabilità dell'algoritmo, proveremo diversi rilanci, e vedremo come degrada la soluzione e come questo influenzi il tempo di esecuzione.

Diverso il caso dell'enumerazione dei potenziali covering cycle. Abbiamo infatti tre diverse euristiche da provare, e tutte con possibili tolleranze da impostare. Poichè a noi non interessa analizzare quanto degradi la soluzione in base alle tolleranze scelte, utilizzeremo le versioni *anytime* dei rispettivi algoritmi. In pratica, se l'euristica termina per una certa tolleranza k entro il tempo limite, viene allora lanciata l'euristica con tolleranza $k+1$, ovviamente con eventuali accorgimenti per evitare di ripetere eventuali soluzioni trovate per la tolleranza precedente, ripetendo il processo finchè non viene raggiunto il tempo limite. Lo scopo di tutto questo è analizzare la scalabilità dei vari algoritmi, e la degradazione della soluzione rispetto all'incertezza dei sensori. Vogliamo inoltre capire quali euristiche funzionano meglio o peggio su quali tipi di grafi, oppure se esiste un'euristica decisamente migliore delle altre, permettendo così di focalizzare le future ricerche alla sua ottimizzazione.

Organizziamo dunque la nostra campagna sperimentale usando

- 10 grafi, a causa dell'elevato tempo di computazione dell'intera campagna sperimentale. Il risultato è, comunque, statisticamente valido;
- la dimensione del grafo presa dall'insieme $\{12, 20, 30, 40, 50\}$, per testare la scalabilità delle euristiche su grafi di grandi dimensioni;
- la densità del grafo pari a 0.25 e a 0.75, come anticipato in Sezione 5.1;
- un'ora come limite temporale massimo di esecuzione per ogni grafo, per ogni α e per ogni rilancio, per questioni pratiche;
- α appartenente all'insieme $\{0.1,0.2,0.3,0.4,0.5\}$. Come già detto in Sezione 5.1, utilizziamo questi parametri per rendere i sensori realistici, dunque evitiamo incertezze troppo alte.;

- il numero di permutazioni eseguite dall'approssimazione dei covset, preso dall'insieme $\{0, 10, 20\}$, per vedere la scalabilità dell'algoritmo e la qualità del preprocessing.

5.4.1 Risultati: risoluzione approssimata del gioco

I risultati dell'esperimento sono visibili nelle Figure 5.7, 5.8, 5.9, 5.10, 5.11, 5.12. Di nuovo, i risultati sono troppo corposi per analizzare tutti gli aspetti contemporaneamente, quindi li osserveremo separatamente. Ricordiamo che se un grafo non viene analizzato (ovvero il preprocessing necessità di più di un'ora di tempo), assume il valore 0, mediato nei grafici.

5.4.1.1 Risultati: scalabilità covset

Com'è possibile notare dai grafici nelle Figure 5.7, 5.8, 5.9, 5.10, 5.11, 5.12, l'algoritmo non scala purtroppo come sperato. Se con 0 rilanci riesce a calcolare i covset di grafi con 30 target (Figure 5.7 e 5.8), già con 10 rilanci e una densità del 75% (Figura 5.10) non termina entra il tempo limite. I grafi da 40 e 50 target, indipendentemente dagli altri parametri, non vengono risolti.

5.4.1.2 Risultati: andamento della soluzione rispetto ai rilanci e all'incertezza

Come si può notare dai grafici nelle Figure 5.7, 5.8, 5.9, 5.10, 5.11, 5.12, il valore della soluzione cresce rispetto al numero di rilanci utilizzati. Questo è ovvio, poichè più i covset si avvicinando ad essere i covset esatti, più la soluzione approssimata tende alla soluzione esatta.

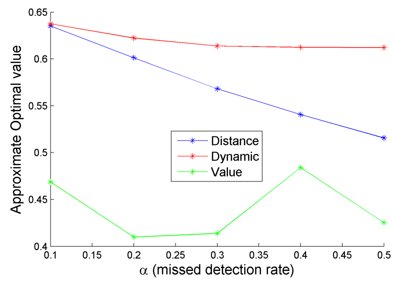
Meno ovvio è, invece, verificare il degrado della soluzione rispetto all'incertezza. Se risolvendo il gioco in modo esatto, è evidente che la soluzione decresca all'aumentare dell'incertezza, risolvendo il gioco in modo approssimato si possono invece avere dei casi particolari nei quali vengono trovati dei particolari covset che permettono di trovare una buona soluzione per un certo α, ma non per un altro. Ad esempio, nelle Figure 5.7(a) e 5.8(a) possiamo vedere come l'euristica dinamica abbia un andamento parabolico rispetto all'incertezza, mentre nelle Figure 5.9(a) e 5.10(a) l'euristica rispetto al valore abbia un andamento quasi casuale. Come già detto, tutto questo si spiega tramite l'utilizzo dei covset approssimati, che possono quindi favorire particolari situazioni rispetto all'algoritmo esatto.

5.4.1.3 Risultati: confronto tra le euristiche

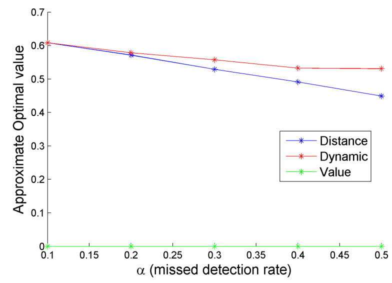
Osserviamo le Figure 5.7, 5.8, 5.9, 5.10, 5.11, 5.12. Tralasciando i grafi con 40 e 50 target, nei quali non è nemmeno stato possibile terminare la generazione dei covset, possiamo notare come in tutti gli altri grafici l'euristica dinamica domini le altre 2, sia in termini di scalabilità che di valore di gioco. In particolare, l'euristica sul valore è quella che si comporta peggio, non riuscendo a terminare nemmeno le computazioni necessarie a partire addirittura dai grafici di 20 target. Inoltre, tra le tre, è l'euristica che fornisce la soluzione peggiore. Questo è probabilmente dovuto allo sfruttamento del solo valore dei target, tralasciando completamente la topologia del grafo, che dunque porta alla ricerca di covering cycle inesistenti, con grandi sprechi computazionali.

L'euristica rispetto alla distanza si comporta bene invece fino ai 20 target, pareggiando spesso l'euristica dinamica quando l'incertezza è bassa, ma purtroppo non riesce a risolvere nemmeno un grafo da 30 target, dove evidentemente la ricerca di sottografi di un particolare diametro è troppo pesante.

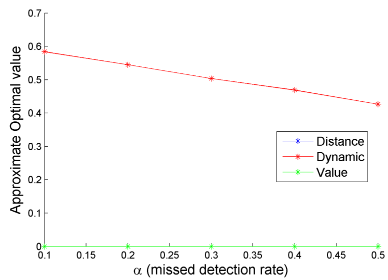
L'euristica dinamica è quindi la migliore. Purtroppo, poichè la generazione dei covset non riesce nemmeno a finire per i grafi da 40 e da 50 target, non possiamo completare le analisi sulla sua scalabilità.



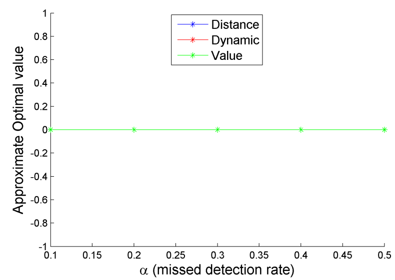
(a) *Grafi a 12 target.*



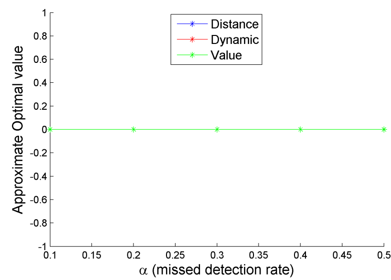
(b) *Grafi a 20 target.*



(c) *Grafi a 30 target.*

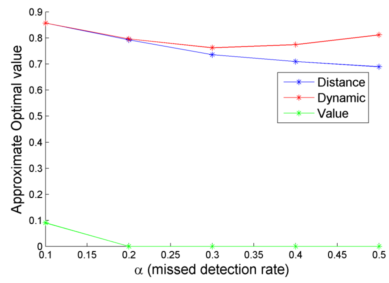


(d) *Grafi a 40 target.*

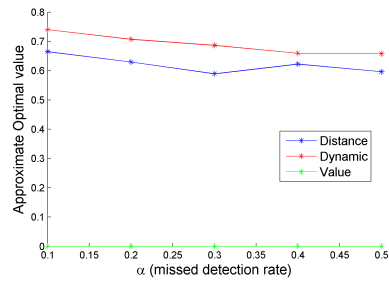


(e) *Grafi a 50 target.*

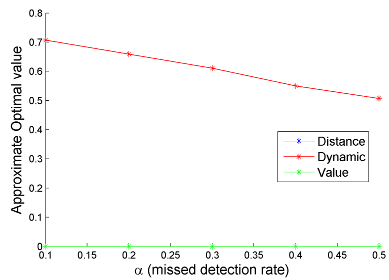
Figura 5.7: Confronto delle euristiche, a seconda dell'incertezza α*, mediato sui grafi con densità del* 25*%, utilizzando* 0 *permutazioni nella creazione approssimata dei covset*



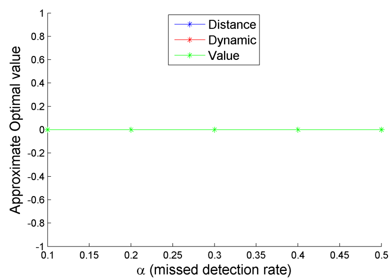
(a) *Grafi a 12 target.*



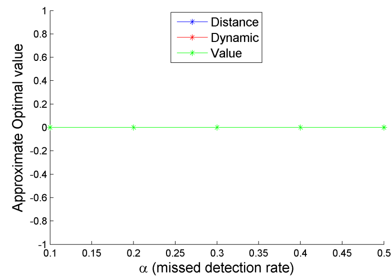
(b) *Grafi a 20 target.*



(c) *Grafi a 30 target.*

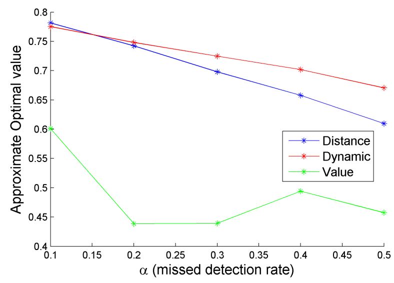


(d) *Grafi a 40 target.*

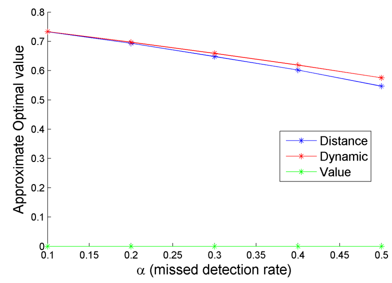


(e) *Grafi a 50 target.*

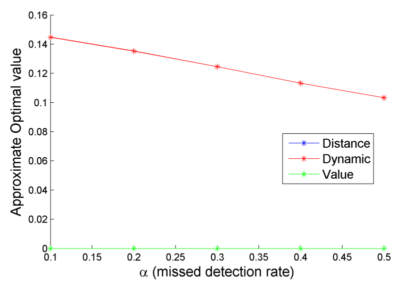
Figura 5.8: Confronto delle euristiche, a seconda dell'incertezza α, mediato sui grafi con densità del 75%, utilizzando 0 permutazioni nella creazione approssimata dei covset



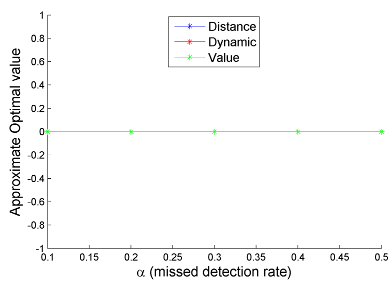
(a) *Grafi a 12 target.*



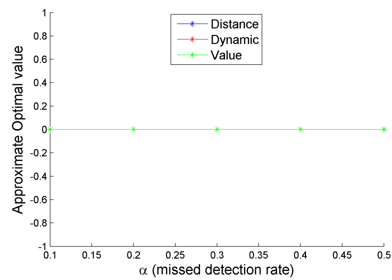
(b) *Grafi a 20 target.*



(c) *Grafi a 30 target.*

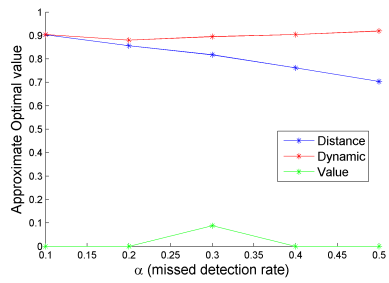


(d) *Grafi a 40 target.*

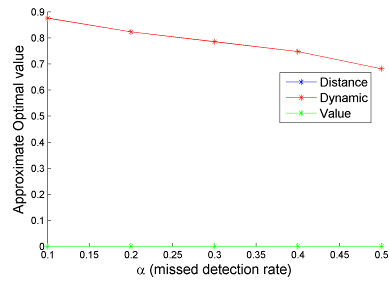


(e) *Grafi a 50 target.*

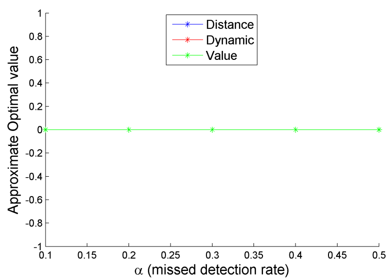
Figura 5.9: Confronto delle euristiche, a seconda dell'incertezza α, mediato sui grafi con densità del 25%*, utilizzando* 10 *permutazioni nella creazione approssimata dei covset.*



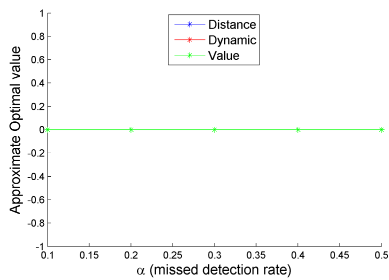
(a) *Grafi a 12 target.*



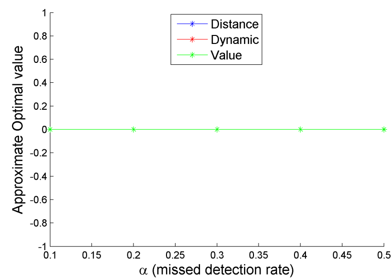
(b) *Grafi a 20 target.*



(c) *Grafi a 30 target.*

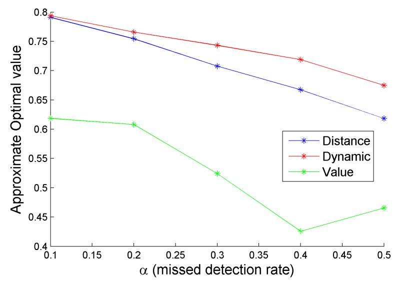


(d) *Grafi a 40 target.*

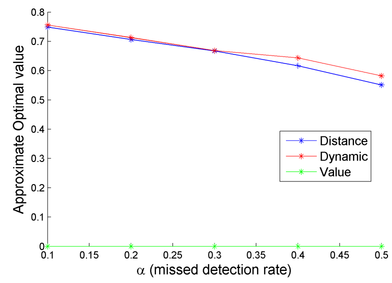


(e) *Grafi a 50 target.*

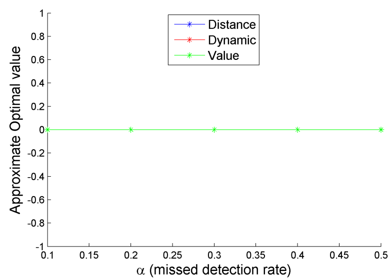
Figura 5.10: Confronto delle euristiche, a seconda dell'incertezza α, mediato sui grafi con densità del 75%*, utilizzando* 10 *permutazioni nella creazione approssimata dei covset.*



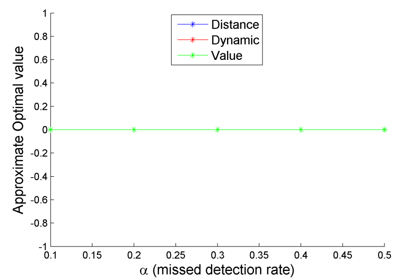
(a) *Grafi a 12 target.*



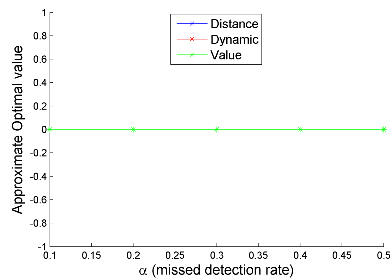
(b) *Grafi a 20 target.*



(c) *Grafi a 30 target.*

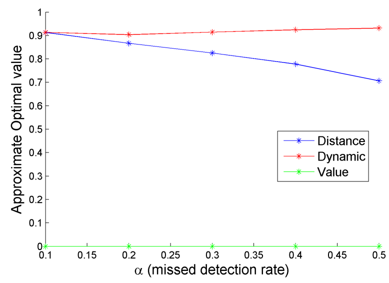


(d) *Grafi a 40 target.*

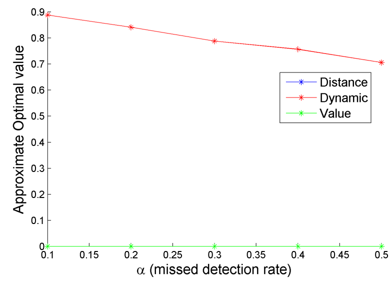


(e) *Grafi a 50 target.*

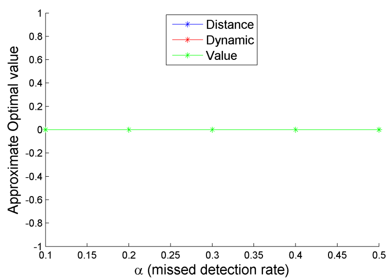
Figura 5.11: Confronto delle euristiche, a seconda dell'incertezza α, mediato sui grafi con densità del 25*%, utilizzando* 20 *permutazioni nella creazione approssimata dei covset.*



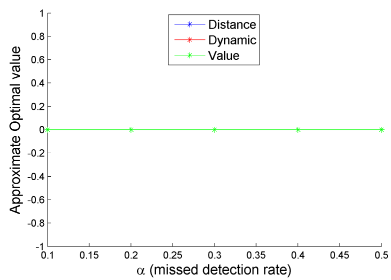
(a) *Grafi a 12 target.*



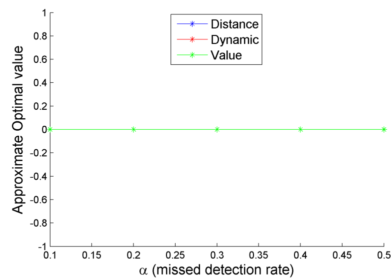
(b) *Grafi a 20 target.*



(c) *Grafi a 30 target.*



(d) *Grafi a 40 target.*



(e) *Grafi a 50 target.*

Figura 5.12: Confronto delle euristiche, a seconda dell'incertezza α, mediato sui grafi con densità del 75%*, utilizzando* 20 *permutazioni nella creazione approssimata dei covset.*

Capitolo 6

Conclusioni e sviluppi futuri

"Cosa aggiungere potrebbe un narratore
a quanto già narrato dall'attore;
a me non resta altro che sparire,
fare un bell'inchino e poi svanire.
Come Cyrano che confessa e muore ai piedi del suo grande eterno amore,
anch'io finito il mio cammino mi accascio e vado verso il mio destino.
Che è quello di chi inizia e già finisce,
sboccia e dopo un attimo appassisce,
di chi vive soltanto un paio d'ore,
sperando in un applauso e dopo muore"

Aldo, Chiedimi se sono felice

Nella seguente sezione, verranno tratte le conclusioni della nostra ricerca, e verranno illustrate possibili aggiunte, estensioni e modifiche al modello da analizzare in futuro.

6.1 Conclusioni

Siamo partiti descrivendo in modo generico i *Patrolling Security Games*. Abbiamo poi analizzato un'estensione particolare, nella quale il difensore ha a sua disposizione un sistema d'allarme spazialmente imperfetto. Abbiamo quindi introdotto nel modello un'incertezza funzionale per i sensori, e abbiamo analizzato, utilizzando negli esperimenti il tempo limite di un'ora, come cambia la risoluzione del gioco, e il suo peso computazionale. Abbiamo testato un algoritmo per la ricerca di un covering cycle, e abbiamo analizzato la sua scalabilità rispetto alla lunghezza massima del ciclo, scoprendo che più aumentano le dimensioni del grafo, più il bound deve essere stringente. Abbiamo provato a risolvere il gioco in modo esatto, scoprendo che è

computazionalmente impraticabile già su grafi di 12 target. Abbiamo quindi presentato delle risoluzioni approssimate, sia per il calcolo dei covering set, sia per l'enumerazione dei possibili covering cycles. Abbiamo scoperto che il calcolo approssimato dei covering set riesce a rendere risolvibili grafi fino a circa 30 target, ma che purtroppo è impraticabile per grafi da 40 target in su. Abbiamo inoltre esaminato i risultati delle tre euristiche per l'enumerazione dei covering cycles, scoprendo che l'euristica dinamica è la migliore sia in termini di valore di gioco sia di scalabilità.

6.2 Sviluppi futuri

La nostra ricerca apre la strada a numerosi sviluppi futuri:

- *Modello multisegnale.* Nei nostri esperimenti abbiamo sempre utilizzato un singolo segnale. Potrebbe essere interessante scoprire come cambiano i risultati e la difficoltà del gioco su grafi grandi riducendo la copertura di ogni segnale ma aumentando il numero di segnali presenti.

- *Modello multidifensore.* Il nostro modello prevede un solo difensore. E' interessante scoprire come cambiano le strategie dei giocatori introducendo più difensori (o più attaccanti), e come varia la complessità computazionale del gioco.

- *Segnali con falsi positivi.* Nel nostro modello abbiamo introdotto i falsi negativi per renderlo più realistico. Per renderlo perfettamente realistico, bisogna introdurre nei segnali anche una certa percentuale di falso positivo (ovvero, l'allarme suona anche se non c'è stato alcun attacco), generalmente più preponderante rispetto a quella di falso negativo.

- *Tuning dell'algoritmo esatto di ricerca dei covering cycle.* Nella nostra ricerca abbiamo provato solo tre possibili bound come lunghezza massima dei covering cycle da trovare su un certo insieme di target. Un ulteriore esperimento potrebbe essere provare più bound, e cercare una relazione tra il bound migliore e la dimensione dei grafi, ad esempio tramite una regressione.

- *Ottimizzazione dell'algoritmo approssimato per la creazione dei covset.* Al momento abbiamo provato ad utilizzare solo 4 ordinamenti, di cui l'ultimo varia randomicamente a seconda del numero di rilanci. Potrebbe essere utile cercare di guidare meglio le permutazioni mischiando i vari ordinamenti, ispirandosi ad esempio ai *genetic algorithms.*

- *Ottimizzazione dell'euristica dinamica.* Abbiamo scoperto che l'euristica migliore è l'euristica dinamica. Bisogna quindi proseguire le analisi e gli studi sulla sua scalabilità, calcolarne la complessità computazionale e cercare, se possibile, di ottimizzare sia l'algoritmo sia l'implementazione.

- *Verificare il funzionamento del modello su casi reali.* Per analizzare l'algoritmo abbiamo utilizzato dei grafi non banali, per poter calcolare realmente il carico computazionale richiesto nel caso peggiore. Generalmente, i grafi reali hanno deadline molto più stringenti, e una densità d'arco molto vicina allo 0, riducendo così drasticamente la difficoltà di risoluzione. E' dunque interessante vedere il comportamento dell'algoritmo su casi reali.

Bibliografia

[1] http://www.imdb.com/title/tt0268978/.

[2] http://www.claymath.org/millennium-problems/millennium-prize-problems.

[3] www.mathworks.com/products/matlab/.

[4] Robert J Aumann. Backward induction and common knowledge of rationality. *Games and Economic Behavior*, 8(1):6–19, 1995.

[5] Theodore Baker, John Gill, and Robert Solovay. Relativizations of the p=?np question. *SIAM Journal on computing*, 4(4):431–442, 1975.

[6] Nicola Basilico, Giuseppe De Nittis, and Nicola Gatti. Adversarial patrolling with spatially uncertain alarm signals. *arXiv preprint arXiv:1506.02850*, 2015.

[7] Nicola Basilico, Nicola Gatti, and Francesco Amigoni. Patrolling security games: Definition and algorithms for solving large instances with single patroller and single intruder. *Artificial Intelligence*, 184:78–123, 2012.

[8] Peter Bürgisser, Michael Clausen, and Mohammad A Shokrollahi. *Algebraic complexity theory*, volume 315. Springer Science & Business Media, 1997.

[9] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[10] George B Dantzig, Alex Orden, Philip Wolfe, et al. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, 5(2):183–195, 1955.

[11] George Bernard Dantzig. *Linear programming and extensions.* Princeton university press, 1998.

[12] Dick Davis et al. *Shahnameh: The Persian book of kings.* Penguin, 2006.

[13] Eddie Dekel and Faruk Gul. Rationality and knowledge in game theory. *Econometric Society Monographs*, 26:87–172, 1997.

[14] David Gale. A theory of n-person games with perfect information. *Proceedings of the National Academy of Sciences of the United States of America*, 39(6):496, 1953.

[15] MR Garey and DS Johnson. Computer and intractability: a guide to theory of np-completness. s. francisco, 1979.

[16] Hsi-Ming Ho and Joel Ouaknine. The cr-uav problem is pspace-complete. *arXiv preprint arXiv:1411.2874*, 2014.

[17] Harold W Kuhn. Extensive games and the problem of information. *Contributions to the Theory of Games*, 2(28):193–216, 1953.

[18] Harold William Kuhn and Albert William Tucker. *Game, Theory of.* Department of Mathematics, Princeton University, 1955.

[19] Howard Kunreuther, Gabriel Silvasi, Eric T Bradlow, Dylan Small, et al. Bayesian analysis of deterministic and stochastic prisoner's dilemma games. *Judgment and Decision Making*, 4(5):363–384, 2009.

[20] Roberto Lucchetti. *A Primer in Game Theory.* Società Editrice Esculapio, 2011.

[21] John Nash. Non-cooperative games. *Annals of mathematics*, pages 286–295, 1951.

[22] John F Nash et al. Equilibrium points in n-person games. *Proc. Nat. Acad. Sci. USA*, 36(1):48–49, 1950.

[23] Christos H Papadimitriou. *Computational complexity.* John Wiley and Sons Ltd., 2003.

[24] Claude E Shannon. Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.

[25] Yoav Shoham and Kevin Leyton-Brown. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations.* Cambridge University Press, 2008.

[26] Albert W Tucker. The mathematics of tucker: a sampler. *Two-Year College Mathematics Journal*, pages 228–232, 1983.

[27] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[28] John Von Neumann and Oskar Morgenstern. Theory of games and economic behavior. *Bull. Amer. Math. Soc*, 51(7):498–504, 1945.

[29] Heinrich Von Stackelberg. *Marktform und gleichgewicht.* J. Springer, 1934.