K-Cavallo che l'erba cresce!

Progetto realizzato da:

Alessio Moretti - Claudio Pastorini

A.A.: 2013/2014

Ingegneria degli Algoritmi

Indice generale

CONTATTI:

Alessio Moretti, 0187698 - alessio.moretti@live.it

Claudio Pastorini, 0186256 - pastorini.claudio@gmail.com

AVVERTENZA: la realizzazione del programma, contenuto nel CD-ROM annesso a tale relazione, è stata implementata in Python 2.7

1. Introduzione

Il progetto “K-cavallo che l'erba cresce!” è un progetto che si basa su una delle più classiche applicazioni nella realizzazione di algoritmi, il gioco degli scacchi. In questo caso però le pedine che si andranno ad adoperare sono solo dei k-cavalli i quali non sono normali cavalli di gioco, ma speciali cavalli che riescono a muoversi più volte nello stesso turno, riescono cioè ad eseguire più salti, più “L” (la mossa del cavallo è la mossa più particolare del gioco degli scacchi, permette al cavallo di saltare letteralmente caselle o altri pezzi facendo idealmente una L sul tavolo da gioco[2]). Il k sta proprio ad indicare quanti salti possano essere effettuati al più in una singola mossa. Cioè se abbiamo un 3-cavallo questo potrà al più compiere tre mosse nello stesso turno, quindi sarà per lui lecito muoversi tre, due o semplicemente una volta nello stesso turno. Dunque un 1-cavallo è un cavallo tradizionale capace di eseguire un solo salto a turno. La richiesta del problema è quella di trovare il minimo numero totale di turni[3] necessari per muovere tutti i k-cavalli nella stessa casa della scacchiera.

Lo stesso problema è noto, in una sua variante, come “Problema di Camelot”. Si dice infatti che Re Artù la vigilia di Natale amasse mettere alla prova i suoi cavalieri giocando su una scacchiera dove un certo numero di cavalli ed un re (presumibilmente lo stesso sovrano anglosassone) dovevano convergere in un punto.

Una particolare chiave di lettura

Se non si è pratici del gioco degli scacchi, di scacchiere e cavalli, questa potrebbe essere una riscrittura del problema:

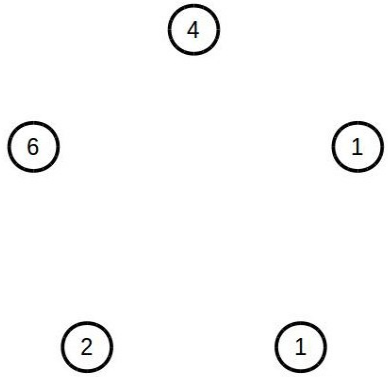
“Ci sono degli amici che vogliono incontrarsi dopo il lavoro per prendersi qualcosa al bar e rilassarsi in compagnia. Lavorano tutti nella stessa città e sono tutti muniti di un qualche mezzo di trasporto. Questi però non hanno tutti la medesima potenza (nel nostro caso quantificabile come velocità per unità di tempo) poiché ci sono biciclette, scooter, auto utilitarie e auto sportive ed è chiaro che chi possiede un'auto sportiva sarà più veloce di chi possiede una bicicletta. Calcolare il minimo tempo possibile che occorre per far incontrare tutti gli amici.”

Il problema ora nasconde dettagli e complicazioni che possono rendere la visione del gioco particolarmente ostica, soprattutto da un punto di vista algoritmico. La cosa che a tutti verrebbe naturale pensare quando bisogna darsi un appuntamento, è quella di incontrarsi a metà strada, ad un bar che si trovi alla stessa distanza da tutti, in un ipotetico centro a cui si possono collegare tutti i punti che rappresentano la posizione di un amico su una mappa. Basterà quindi contare quanto occorre ad ogni amico per raggiungere quel punto e otterremmo così il tempo necessario per farli arrivare tutti allo stesso bar. Così facendo però non troveremo il minimo tempo perché non andiamo a sfruttare a pieno le peculiarità dei mezzi. Dobbiamo quindi riconsiderare la scelta del bar e sfruttare al massimo le risorse.

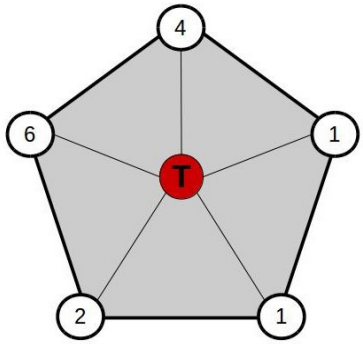
2 Nel gioco degli scacchi, il cavallo si muove di due passi in orizzontale (verticale) seguito da un passo in verticale (orizzontale), in modo che il tragitto percorso formi idealmente una "L". Ovvero, se un cavallo si trova nella posizione (casa) (x, y) della scacchiera, esso può saltare in una delle seguenti otto case: (x+1, y+2), (x+1, y-2), (x+2, y+1), (x+2, y-1), (x-1, y+2), (x-1, y-2), (x-2, y+1), (x-2, y-1), ovviamente senza uscire dalla scacchiera.

3 Si è deciso di cambiare la terminologia usata nel testo del problema per rendere questo di più immediata comprensione e per attenersi maggiormente alla terminologia di tipo scacchistico. Sono stati sostituiti quindi i termine “salto” del testo del problema con “mossa” e “mossa” con “turno”.

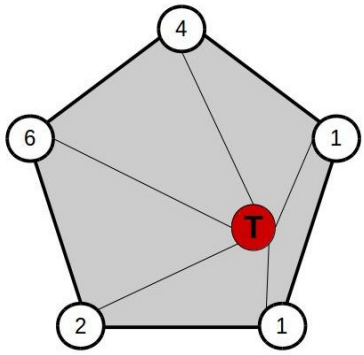
Facciamo un esempio:



In questo caso abbiamo i nostri cinque amici che sulla mappa vanno a formare un pentagono. Ogni amico è rappresentato da un punto al cui all'interno vi è un numero che indica la velocità del mezzo per unità di tempo. Possiamo allora pensare di sfruttare queste informazioni per ricalcolare il punto dell'incontro. Consideriamo quindi il centro di massa della distribuzione degli amici, un punto puramente geometrico che si adopera in fisica per studiare sistemi di punti materiali. Quindi se nel nostro caso il centro del pentagono è:



quella del centro di massa dello stesso sarà invece:



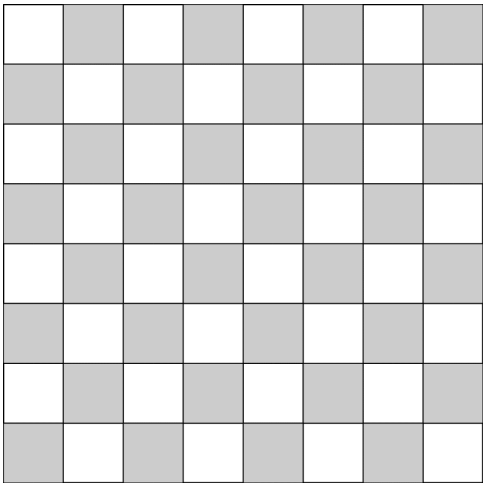
grazie alla seguente formula (invertendo i valori in modo tale da dar più peso ai valori più piccoli):

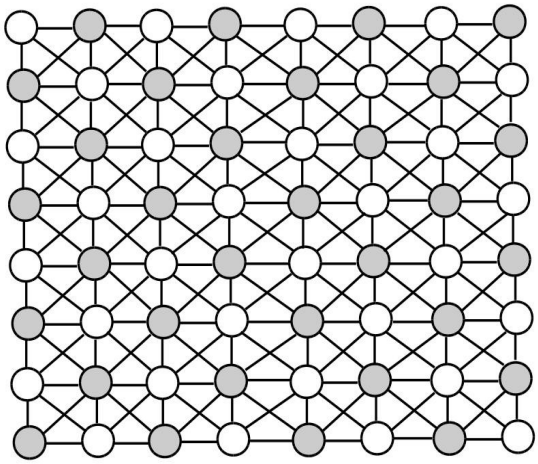
$$r_{cm} = \sum_{i=0}^{n} \left(\frac{m_i \cdot r_i}{m_i} \right)$$ (con r vettore posizione dall'origine e m massa)

Come possiamo vedere è puramente una media pesata delle posizioni ed è proprio quello che cercavamo, trovare una posizione media che però andasse a valutare anche la potenza dei mezzi dei nostri amici. Abbiamo quindi trovato un modo per risolvere il problema e abbiamo tutti i nostri amici allo stesso bar nel minor tempo possibile.

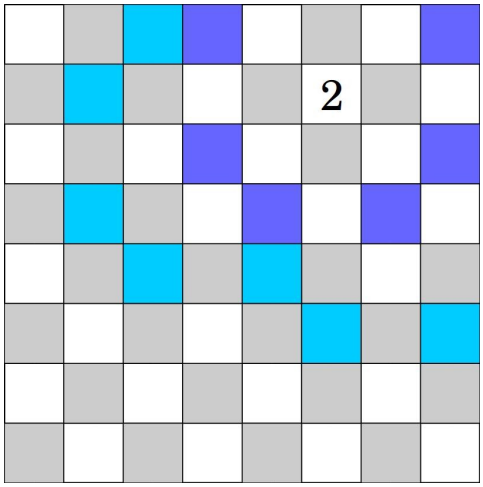
Un problema ben noto

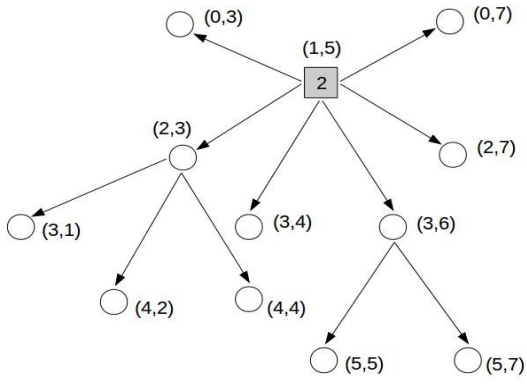
Da un punto di vista formale il problema può essere formulato basandoci sulla teoria dei grafi. Quindi spendendo un poco di energie in più è possibile, sfruttando abilmente la semplice funzione che mette in relazione la posizione attuale di un cavallo con le sue possibili mosse, costruire un grafo della scacchiera e dei suoi k-cavalli.





Nel caso di un 2-cavallo, ad esempio, osserviamo come possiamo riformulare il grafo generico della scacchiera come un grafo costruito sulle mosse del cavalli [4] (idealmente estendendo il ragionamento fino alla costruzione di un intero “Knight's Tour” - anche questo problema concettualmente molto noto):





4 Verranno rappresentate solo alcune mosse per non appesantire le illustrazioni

Abbiamo così portato un problema del mondo fisico in un problema di tipo informatico la cui natura non è di certo aliena ad una persona che abbia familiarità con i grafi e le loro analisi. Possiamo quindi risolverlo grazie ad un set di algoritmi di cui analizzeremo in apposita sede il costo computazionale.

2. Approcci implementativi

analizziamo diverse soluzioni di natura algoritmica

Se Maometto non va alla montagna...

Mostriamo qui come risolvere il problema dopo aver utilizzato la chiave di lettura proposta in introduzione. Quello che andremo a fare è la riformulazione in termini del problema originario della soluzione trovata in precedenza con il problema degli amici e del bar. Gli amici con i loro mezzi con potenza k saranno i k-cavalli. Questi non risiederanno più nella stessa città ma in una stessa scacchiera. Quello da trovare non sarà più il bar che permette di essere raggiunto nel minor tempo possibile ma la casa della scacchiera che permetterà di far incontrare tutti i k-cavalli nel minor tempo possibile.

(problemi di trasporto)

Questo primo approccio fa quello detto in precedenza. Si calcola la casa destinazione e vi si fanno tendere tutti i cavalli verso questa casella fino a quando non ci si troveranno tutti. Un simile approccio se può sembrare a voce di facile esposizione non risulta banale da implementare per via algoritmica. Il problema che si presenta di maggiore difficoltà è quello di calcolare una corretta casa target e di scegliere la mossa da fare eseguire ogni qualvolta il cavallo non riesce a con una o con k mosse a raggiungere il target. Vediamo perché.

Per calcolare la casa target abbiamo detto che andremo a calcolare un baricentro e lo facciamo nel seguente modo:

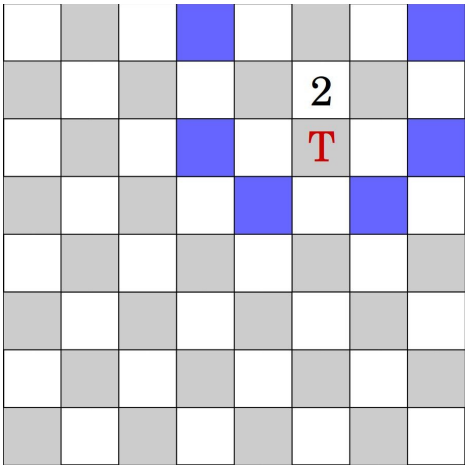
```
1. def calculate_target(match):
2.     m = 0
3.     mr_row = 0
4.     mr_col = 0
5.     pieces = match.get_knights()
6.     max_value = match.get_max()
7.     num_pieces = len(pieces)
8.
9.     if num_pieces > 1:
10.         for piece in pieces:
11.             if max_value != piece.get_value():
12.                 value = abs(piece.get_value() - max_value + 1)
13.             else:
14.                 value = piece.get_value() - max_value + 1
15.             m += value
16.             mr_row += piece.get_row() * value
17.             mr_col += piece.get_col() * value
18.
19.         target_row = round_int(mr_row / m)
20.         target_col = round_int(mr_col / m)
21.
22.         return (target_row, target_col)
23.
```

```
24.     elif num_pieces == 1:
25.         return 0
26.     else:
27.         return -1
```

Passando come argomento un match[5] alla funzione calculate_target() questa restituirà nel caso sia possibile una tupla contenente la posizione riga, colonna della casella destinazione calcolata secondo la formula vista in precedenza del centro di massa (con varie accortezze[6]) e 0 nel caso non sia possibile e questo accade se vi è presente o solo un pezzo o nessuno sulla scacchiera. All'interno di questa funzione viene richiamata una funzione chiamata round_int() che permette di arrotondare per eccesso o per difetto un numero passato per argomento:

```
1. def round_int(number):
2.     if trunc(number + 0.5) > trunc(number):
3.         return trunc(number) + 1
4.     else:
5. return trunc(number)
```

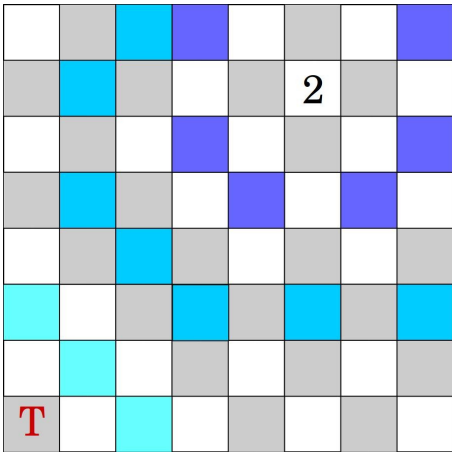
Iniziano qui i primi problemi poiché essendovi una divisione si viene a creare per forza di cose un valore frazionario e sappiamo tutti che le case delle scacchiere sono elementi discreti e che quindi non possono essere rappresentati da numeri reali. Proprio per questa è stata creata una funzione apposita per l'arrotondamento, anche così facendo però non si trova veramente la destinazione, ma un'approssimazione che si spera essere la reale destinazione. Potremmo però dopo ogni mossa di tutti i cavalli che tendono a questa casella aggiornare nuovamente la casella destinazione in modo tale da tendere al limite alla casella corretta. Il secondo problema in cui ci siamo imbattuti e non ci ha permesso di implementare in modo soddisfacente questo approccio è la caratteristica principale del cavallo, il suo movimento. Come spiegato all'inizio il cavallo si muove formando idealmente una L sul tavolo di gioco. Così facendo vediamo forma idealmente un ottagono e può raggiungere solo i vertici di questo. Non può raggiungere, almeno non banalmente, i punti all'interno dell'ottagono e questo crea parecchi problemi. Si prenda ad esempio un cavallo che si trovi in questa situazione:



5 Non verranno discusse in questa sezione le strutture dati utilizzate con i propri attributi o metodi, per un maggior approfondimento rimandiamo alla lettura dell'appendice A

6 Le accortezze riguardano il cambiamento dei pesi dei cavalli. Abbiamo dovuto fare una simile operazione perché la formula del baricentro valuta i pesi maggiori e non i minori come noi vorremmo!

Che cosa occorre fare per far muovere il cavallo nella casella target (contrassegnata in figura con una T)? Bisogna far allontanare il cavallo della distanza tale che gli permetta poi con un'ulteriore mossa di raggiungere la casella T. È molto complicato fare questo come è molto complicato scegliere una casella in cui far andare un cavallo quando questo non cade con nessuna delle sue k-mosse nel target:



Anche qui, che si può fare? La logica ci direbbe di andare a muovere il cavallo in modo tale da avvicinarsi il più possibile alla casella target, ma non troppo, altrimenti si cadrebbe nel problema di prima. Questi due problemi comportano una accortezza tale che ci ha fatto desistere nel completare l'implementazione di questo approccio[7]. Una soluzione semplice, ma che ci è parsa poco fattibile, è stata quella di scegliere mosse a caso per allontanarsi od avvicinarsi al target. Questo però non permetteva più di essere sicuri di trovare il minimo numero di turni, si è pensato quindi di fare un numero maggiore di prove e di prendere il minimo, ma anche qui la scelta non era molto praticabile dato che i cavalli potevano essere molti e comportavano il ricalcolare tutto ogni volta: avere un approccio così “naif” non ci piaceva. Ma se Maometto non va alla montagna...

...la montagna va da Maometto!

Il titolo ci è parso molto azzeccato per queste proposte e speriamo che vi si abbia già fatto intuire il motivo!

Il problema che incontravamo era quello di far muovere i cavalli? Abbiamo quindi scelto di non farli muovere più e di “muovere” la casella target. Abbiamo quindi creato un finto cavallo sulla casella target e abbiamo calcolato le sue mosse finché non fosse possibile raggiungere tutti i cavalli.

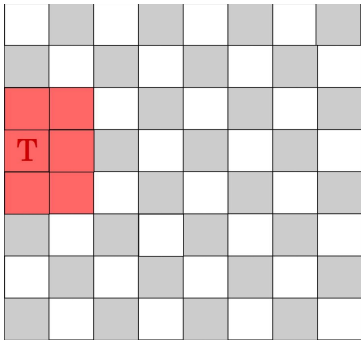
Per calcolare infine il numero di turni abbiamo contato quanta era la distanza che c'era tra ogni cavallo e il target e abbiamo quindi calcolato a seconda del valore di k i turni necessari. Vediamo il codice:

```
1. targets = calculate_targets(match)
2.     if isinstance(targets, list):
3.         turns = float('inf')
4.         for target in targets:
5.             target_knight = Knight(target[0], target[1], 1)
6.
7.             force = False
8.
```

7 Non avendo completato l'implementazione in modo soddisfacente non proponiamo tale approccio come soluzione del problema, per questo non discutiamo interamente il codice e non lo alleghiamo ma lo presentiamo come evoluzione delle nostre idee

```
9.              while not match.is_finished():
10.                 list_moves = target_knight.get_moves(match.get_rows(),
                                match.get_cols(), debug) if target_knight.get_value() ==
                                1 else target_knight.get_other_moves(match.get_rows(),
                                match.get_cols(), debug)
11.                 if len(list_moves) != 0:
12.                     for knight in match.get_knights():
13.                         if target == knight.get_position() and not
                                        knight.is_founded():
14.                             match.knight_found(knight, 0)
15.                         for move in list_moves:
16.                             if move == knight.get_position() and not
                                        knight.is_founded():
17.                                 match.knight_found(knight, target_knight.get_value())

18.                 else:
19.                     force = True
20.
21.                 try:
22.                     match.finish(force)
23.                 except Exception:
24.                     pass
25.
26.                 target_knight.set_value(target_knight.get_value() + 1)
27.             if match.get_turns() < turns:
28.                 turns = match.get_turns()
29.
30.             match.reset()
31.         if turns != float('inf'):
32.             print("Ho impiegato " + str(turns) + " turno(i)!")
33.         else:
34.             print("Impossibile!")
35.     else:
36.         if targets == 0:
37.             print("Ho impiegato 0 turni!")
38.         else:
39.             print("Impossibile")
```



Come si può intuire è stata usata la funzione calculate_targets(), una versione modificata della funzione calculate_target(), per calcolare una lista, e non più una singola tupla, di targets. Questo è stato fatto per ovviare al problema dell'approssimazione di cui abbiamo già discusso in precedenza. La funzione ora non restituisce solo la tupla contenente la posizione del target ma una lista di posizioni. Si è preso il punto calcolato ed un suo intorno. Quindi con la nuova funzione se la casa target calcolata è (3, 0)[8] si prenderanno anche: (2, 0), (2, 1) etc.

8 La convenzione che è stata usata per descrivere le posizioni sulla scacchiera è la seguente: (numero riga, numero colonna), con i

Dopo aver calcolato quindi le case da valutare per ogni casella si crea un cavallo fake e si calcolano le sue mosse finché non incontra tutti i cavalli sulla scacchiera. Una volta trovati si verifica che i turni che ha impiegato sono minori di quelli impiegati precedentemente utilizzando target (e quindi cavalli fake) differenti. È stato fatto uso per trovare il minimo della funzione float() passandogli come parametro il termine inf. Così facendo verrà restituito il valore più grande rappresentabile in Python, che, per i nostri scopi, equivale ad un $+\infty$. In questo modo siamo sicuri che facendo il primo confronto il risultato sia corretto senza dover tentare la sorte inizializzando la variabile turns ad un valore elevato. Per il resto il codice parla da sé, sono stati usati i metodi delle classi Match e Knight e quindi si nascondono la maggior parte dei particolari, ma è comunque comprensibile il codice sopra riportato.

Analizziamo matematicamente il tempo computazionale, assumendo costante il calcolo del target:

$$T_{montagna}(n) = T_{target}(n) + t(p) \cdot T_{match}(n)$$

Detto t il numero di target calcolati, sperimentalmente si è visto che l'affidabilità della soluzione[9] *è proporzionale a t che è in funzione della precisione. In particolare, all'aumentare di questa, diminuisce t.*

Si ha quindi, per n numero di mosse totali eseguibili, per k numero di cavalli e per h massima distanza percorribile dal cavallo prima di uscire dai limiti della scacchiera (o richiudere il cammino su se stesso):

$$T_{montagna}(n) = O(k) + t(p) \cdot \sum_{i=0}^{h} \alpha^i \cdot O(1) = O(k) + t(p) \cdot O\left(\frac{\alpha^{h+1}-1}{\alpha-1}\right) = O(k) + t(p) \cdot O(n)$$

Sulle spalle dei giganti

In questo approccio ci si ricondurrà ad un grafo delle mosse del cavallo, che come vedremo sarà possibile considerare tanto collo di bottiglia nell'analisi dei tempi di esecuzione, quanto punto di forza dell'analisi delle mosse.

Definendo il problema come un grafo possiamo passare ad analizzare il problema del minimo numero di mosse con uno degli algoritmi per lo studio dei cammini minimi a partire da una sorgente singola. Le sorgenti saranno proprio le posizioni iniziali dei cavalli ed il grafo generato sarà formato dai loro cammini possibili, costruiti di modo tale da impedire l'instaurarsi di cammini chiusi rispetto ad uno stesso cavallo.

L'intero approccio, per mantenere una certa modularità delle implementazioni, sarà basato su una struttura orientata agli oggetti, basata su tre classi fondamentali: Match, Knight e Graph.

Match verrà utilizzato principalmente per fornire un'interfaccia per:

- inizializzare l'istanza del problema a partire dall'input mantenendo informazioni accurate sul numero dei pezzi, sulle dimensioni della scacchiera, sulle posizioni iniziali dei cavalli etc.;
- costruire il grafo delle mosse ed accedervi per poter lavorare sugli algoritmi di ricerca dei cammini minimi;

valori che partono da 0 e arrivano ad n-1 (con n numero righe/colonne totali). Quindi in questo caso la casa contrassegnata con T nella posizione (3, 0) non è altro che la casa A5 nel gergo scacchistico

9 Per valutare l'affidabilità della soluzione ci si affiderà nei capitoli successivi alle evidenze sperimentali

- gestire gli algoritmi di ricerca dei cammini minimi e ricavarne dai risultati la soluzione al problemi.

Knight è la rappresentazione tramite oggetto del cavallo, quindi mantenente le informazioni:

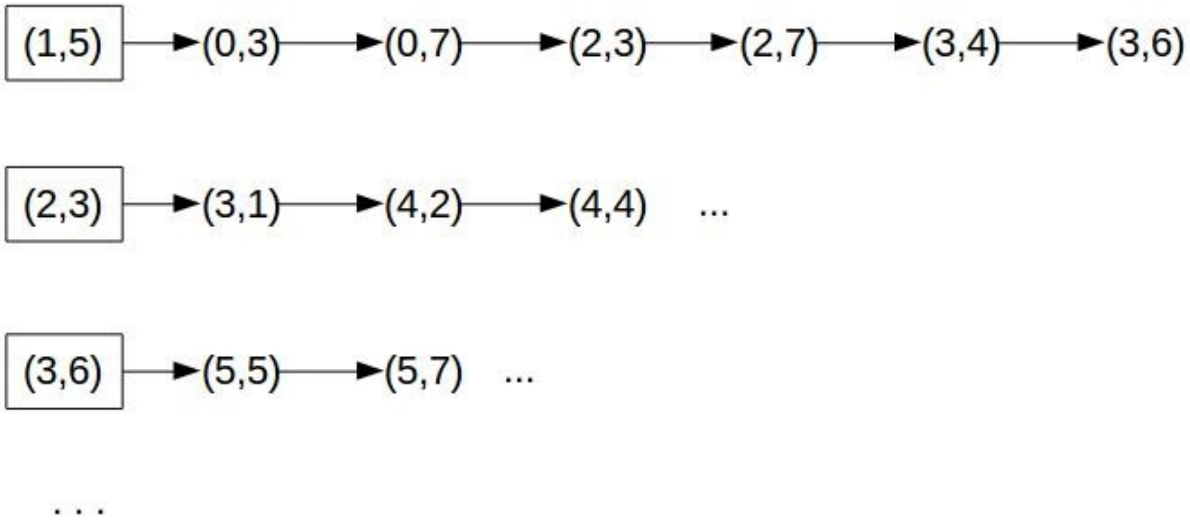
- posizione del cavallo;
- valore del cavallo indicante le mosse possibili in un turno;
- mosse calcolate.

e performante le seguenti operazioni:

- singleMove: calcolo di un singolo set di mosse di un cavallo a partire da una certa posizione relativamente ai limiti imposti dalla scacchiera;
- completeTour: calcolo di un completo tour delle mosse del cavallo a partire dalla sua posizione iniziale, espandendo quindi dinamicamente la lista delle possibili mosse.

Preludio alla Fondazione: grafo delle mosse

La rappresentazione scelta per questo grafo delle mosse è quella delle liste di adiacenza. Rispetto all'esempio del grafo visto precedentemente avremo quindi:



Quindi la classe Graph manterrà due dizionari, uno inizializzato con arbitrari indici e contenente i nodi, l'altro che permetterà di mantenere tramite gli stessi indici una lista di adiacenza, codificata come lista doppiamente collegata per aumentare l'efficienza delle operazioni di inserimento e le eventuali operazioni di cancellazione.[10]

I nodi sono a loro volta codificati da una particolare classe Node che manterrà, oltre al proprio indice e

10 Questa considerazione deriva dall'iniziale necessità di garantire una soluzione modulare e facilmente adattabile alle esigenze progettuali. Andando avanti con la fase di ingegnerizzazione si è man mano ridotta tale modularità in virtù di accorgimenti tecnici, quali lo sfruttamento di buffer riempiti in fase di inizializzazione, come lo sfruttamento della velocità di accesso in memoria garantita dall'implementazione nativa delle liste in Python, etc.

l'elemento conservato, anche un certo “peso” che identificherà nella lista di adiacenza la presenza o meno di un cavallo nella posizione. Per permettere di tenere traccia delle mosse dei cavalli durante gli algoritmi di visita e di ricerca dei cammini minimi, la classe che rappresenta il generico nodo del grafo manterrà anche le informazioni seguenti:

- numero dei cavalli che raggiungono il nodo;
- peso totale degli archi per raggiungere il nodo.[11]

Procediamo quindi a descrivere la creazione del grafo a partire da un generico oggetto di tipo Match di cui analizzeremo il metodo makeGraph():

```
1. def makeGraph(self):
2.       r = self.get_rows()
3.       c = self.get_cols()
4.
5.       for k in self._knights:
6.             kgt = self.setGraph().insertNode(k.get_position(), k)
7.             self._knights_nodes.append(kgt)
8.             k.completeTour(r, c)
9.
10.      for knight in self._knights:
11.            for step in knight.getMoves():
12.                  move_from = step[0]
13.                  move_to = step[1]
14.                  node = self.setGraph().insertNode(move_from)
15.                  moveNode = self.setGraph().insertNode(move_to)
16.                  self.setGraph().linkNode(node, moveNode)
17.            knight.refreshBuffer()
```

Vengono quindi per prima cosa inseriti nel grafo relativo al match (richiamato tramite un opportuno metodo di interfaccia) i nodi contenenti i cavalli nelle loro posizioni iniziali e nello stesso ciclo vengono calcolate tutte le possibili mosse degli stessi cavalli.

Nel secondo ciclo, per ogni cavallo vengono esaminate le mosse. All'interno dell'oggetto Knight le mosse sono conservate in una lista (built-in Python) in forma di tupla come segue: (position_from, position_to). Inoltre il metodo insertNode, nel caso in cui il nodo relativo alla posizione sia già effettivamente presente[12], restituisce il nodo con il suo peso specifico (questi vale 0 se nessun cavallo è presente, altrimenti vale proprio un'istanza della classe Knight, giusto a titolo esemplificativo di chiarezza strutturale), permettendo con una semplice scansione delle liste di inizializzare i nodi e le loro liste di adiacenza. Infine puliamo il buffer delle mosse utilizzato durante la fase di calcolo delle mosse.

Questa strutturazione permette di fare al più n inserimenti dei nodi per un singolo cavallo. Se tutti i nodi sono inseriti dopo il primo calcolo delle mosse[13], allora si provvede unicamente a collegare i nodi nelle liste di adiacenza al più m volte, codificando quindi, anche se non esplicitamente, gli archi del grafo ed evitando, per ovvie ragioni, di inserire duplicati nelle liste di adiacenza.

11 Il calcolo verrà effettuato da una apposita funzione codificata come metodo della classe Knight, descritta in Appendice A. Come si vedrà tale esigenza nasce come sempre dall'alta modularità dell'istanza iniziale del grafo.
12 Tale controllo è ottimizzato, sfruttando l'implementazione nativa in Python della lista che funge da buffer delle mosse.
13 Tale controllo viene effettuato sui dizionari, implementati nativamente, attributi del grafo delle mosse che è, in questo momento, in fase di creazione.

A questo punto ci soffermiamo ad analizzare il metodo della classe Match che permette la costruzione del grafo: questa procede sull'idea di estendere il fronte delle possibili mosse del cavallo, già calcolate, con un nuovo calcolo delle mosse rispetto ad ogni punto raggiunto dal fronte.

```
1. def completeTour(self, rows, cols):
2.      moves = [self.get_position()]
3.      start = 0
4.      stop = len(moves)
5.      while start != stop:
6.           for move in moves[start:stop]:
7.                new_moves = self.singleMove(move, rows, cols)
8.                for new in new_moves:
9.                    self._moves.append((move, new))
10.               moves += new_moves
11.          start = stop
12.          stop = len(moves)
```

In particolare, noto che per ogni posizione occupata, effettivamente o virtualmente, da un cavallo su una scacchiera sono possibili al più 8 mosse: considerando ognuna di queste possibili mosse come posizione raggiunta dal cavallo e ricalcolando da essa un nuovo set di mosse, è possibile costruire una lista di tuple che, come abbiamo visto, verranno usate per il linking dei nodi del grafo.

Analizziamo matematicamente il costo computazionale, assumendo che l'inizializzazione di una mossa è eseguita in tempo costante, avremo in generale, definito h *come massimo livello raggiunto generando nuove mosse.*

$$T_{completeTour}(n) = \sum_{i=0}^{h} \alpha^i \cdot O(1) \rightarrow T_{completeTour}(n) = O(\frac{\alpha^{h+1}-1}{\alpha-1}) \rightarrow O(n)$$

dove, al più, α *è pari proprio ad 8, come visto prima. Quindi considerando il numero dei nodi pari proprio al valore dato dalla serie geometrica visto sopra e noto che il numero degli archi non può essere più di* α *volte il numero dei nodi, possiamo analizzare il tempo totale, nel caso peggiore, per:*

$$T_{makeGraph}(n) = k \cdot T_{completeTour}(n) + k \cdot T_{graphInitialization}(n) = k \cdot O(n) + k \cdot O(m+n) = k \cdot O(m+n)$$

e tenendo conto che il grafo è rappresentato da liste di adiacenza, possiamo fornire una stima anche dello spazio di memoria occupato in:

$$S_{graph}(n) = O(m+n)$$

In ultima analisi per uno spazio occupato, lineare rispetto al numero di nodi e di archi, abbiamo un tempo di costruzione del grafo pari ad un valore proporzionale al numero dei cavalli del problema e alla grandezza del grafo. Garantiamo quindi una copertura completa di tutte le possibili mosse, inizializzando opportunamente i nodi del grafo e mantenendo le informazioni sui percorsi possibili codificandole nelle liste di adiacenza.

La costruzione del grafo è quindi per ora sia un tallone d'Achille rispetto al costo computazionale, quanto si dimostrerà un formidabile strumento di analisi rispetto ai cammini minimi[14].

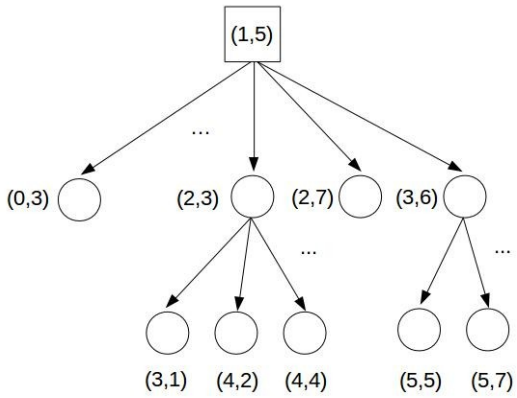
14 Come si vedrà più avanti, la costruzione del grafo è ulteriormente ottimizzabile, anche se per ora ci baseremo su una classica rappresentazione del grafo delle mosse e cercheremo di studiare alcune strategie convincenti di ricerca di cammini minimi a

Prima Fondazione: Breadth-First Search

Arrivati al punto di avere un grafo delle possibili mosse, è bene sottolineare che questo vuol dire implicitamente avere indicazioni su tutte le possibili posizioni sulla scacchiera, ora codificate come nodi del grafo, raggiungibili dai cavalli. Di conseguenza è stato definito implicitamente il grafo delle raggiungibili posizioni.

E' inoltre particolarmente importante un'osservazione che nella creazione del grafo è stata trascurata: gli archi del nostro grafo, codificati nelle liste di adiacenza, sono tutti di costo unitario! Questa osservazione permette di svincolare il cavallo dalle sue proprietà di k-cavallo[15], salvo poi valutarla in sede di analisi dei risultati ottenuti dall'algoritmo di visita che stiamo per adottare.

Infatti, osserviamo che per un grafo i cui archi sono di costo unitario, un algoritmo di visita in ampiezza può produrre un risultato notevole...



Otteniamo di fatto, a partire da un certo nodo, un albero che rappresenta tutti i possibili cammini minimi a partire da quel determinato vertice!

Questo è noto generalmente nelle teoria dei grafi, rispetto l'algoritmo di visita in ampiezza, enunciando e ricavando il seguente lemma:

“Siano dati un grafo orientato e connesso G = (V, E), con V insieme dei vertici ed E insieme degli archi, un vertice s in G ed un albero T prodotto dall'algoritmo di visita in ampiezza a partire dal nodo s. Per ogni vertice v risulterà $l(v) = d_{sv}$ ”

In altre parole, interessandoci dei soli archi in avanti prodotti dalla visita, possiamo affermare che: *“i vertici dell'albero vengono generati il più vicino possibile alla radice”*.

Oltretutto risulta conveniente, vista la codifica del grafo nella nostra implementazione tramite liste di adiacenza, utilizzare un algoritmo di visita per poter mantenere un tempo di esecuzione proporzionale al numero di nodi ed archi, rispetto al numero di cavalli da cui far partire la visita. Questo ci ricondurrà ad un risultato paragonabile a quello ottenuto per la creazione del grafo stesso, tranne ovvie costanti moltiplicative che, nascoste dalla notazione asintotica, si fanno sentire su grandi iterazioni.

partire da soluzioni algoritmiche ben note nella teoria dei grafi.

15 Questa decisione è stata presa per poter lavorare il più semplicemente possibile sulla visita, senza vincolare tale visita al cavallo stesso, almeno durante la sua esecuzione. Tale lavoro può essere recuperato comunque sia in tempo lineare.

Quindi, conservando nelle foglie la distanza dalla radice dell'albero e conservando le foglie stesse, per semplicità implementativa come si vedrà a breve, in una semplice coda, codificata a partire dalla classe delle liste doppiamente collegate, possiamo formulare un algoritmo di visita in ampiezza. La coda infatti manterrà l'informazione sull'ordine di visita, permettendo una scansione in ampiezza di tutti i figli di un vertice prima di passare ad analizzare i figli dei vertici successivi.

Implementando la visita come metodo della classe Graph avremo quindi:

```
1. def visitBFS(self, node):
2.       unexplored = 0
3.       explored = 21
4.       closed = 42                          #So long and thanks for all the fish!
5.       self.validateNodes(unexplored)
6.       node.set_token(explored)
7.       T_root = Leaf(node)
8.       T_root.setDistance(0.0)
9.       T = Tree(T_root)
10.
11.      F = Queue()
12.      F.enqueue(T_root)
13.
14.      while not F.isEmpty():
15.            u = F.dequeue()
16.            n = u.getElem()
17.            n.set_token(closed)
18.            for v in self._adjacency[n.get_index()].getLastAddedList():
19.                if v.get_token() == unexplored:
20.                    v.set_token(explored)
21.                    l = Leaf(v)
22.                    F.enqueue(l)
23.                    T.insertLeaf(l, u)
24.      return T
```

L'algoritmo marca inizialmente tutti i nodi come inesplorati ed inizializza un albero T contenente il solo vertice iniziale, per l'esattezza uno dei vertici contenenti un cavallo. Quindi viene inizializzata una frangia F in cui mantenere le foglie contenenti i nodi esplorati ma non ancora completamente visitati, cioè quelli di cui si deve procedere ad esaminare la lista di adiacenza corrispondente. Quando questa viene esaminata, se i nodi in essa contenuti sono ancora inesplorati, vengono marcati come esplorati e vengono inseriti in coda, quindi vengono resi figli del nodo da cui sono stati per la prima volta raggiunti. Il metodo insertLeaf permette di mantenere le distanze conservando la distanza della nuova foglia come distanza del padre più un valore unitario. Una volta che un nodo esplorato viene estratto della coda come contenuto di una foglia già inizializzata, fino a che la coda non risulti come vuota, viene marcato come chiuso.

Viene quindi fatto ritornare l'albero T così generato. In particolare un metodo più generico della classe Graph permette di recuperare una lista di alberi, di dimensione pari esattamente alla lista dei cavalli iniziali. A partire da questa “foresta” di alberi BFS una semplice scansione delle foglie permette di tenere traccia dei nodi visitati sfruttando gli attributi già precedente descritti.

In particolare per ognuna delle foglie, verrà prelevata la distanza e ricalcolata in base al massimo numero di mosse in un turno che possono essere effettuate dal cavallo. Quindi questo peso verrà conservato nel nodo stesso come distanza e la minima somma di tale distanze, per ognuno dei nodi raggiunti da tutti i cavalli, sarà

pari proprio al minimo numero di mosse per far convergere i pezzi sulla scacchiera in un'unica posizione.

Dimostriamo matematicamente i tempi di esecuzione, assumendo come al solito costanti gli accessi in memoria, abbiamo:

$$T_{validation}(n) = n \cdot O(1) = O(n)$$

poiché il passaggio da inesplorato ad esplorato può avvenire al più una volta e possono esserci al più m *operazioni di aggiornamento della frangia:*

$$T_{fringe}(n) = O(m+n)$$

quindi l'esplorazione può avvenire al più n volte per esaminare i nodi ed m volte per esaminare le loro liste di adiacenza:

$$T_{visitBFS}(n) = T_{validation}(n) + T_{fringe}(n) + O(m + n) = O(m + n)$$

a questo punto non rimane che iterare per il numero dei cavalli l'algoritmo e recuperare i risultati; sempre considerando in tempo costante sia l'accesso in memoria che la funzione di calcolo delle distanze e proporzionale al numero dei nodi la funzione di recupero delle informazioni dalla visita:

$$T_{visitNodesBFS}(n) = T_{makeGraph}(n) + k \cdot T_{visitBFS}(n) + k \cdot O(n) = k \cdot O(m + n)$$

L'algoritmo di visita in ampiezza viene quindi eseguito proporzionalmente al numero di nodi, archi e cavalli ed in un tempo confrontabile, a meno di costanti moltiplicative nascoste dalla notazione asintotica, al tempo di creazione del grafo, permettendoci di mantenere il tempo computazionale nella linearità.

Seconda Fondazione: algoritmo di Dijkstra

Ottenuto un grafo delle possibili mosse, potrebbe essere interessante, nonostante i risultati positivi ottenuti con l'algoritmo di ricerca in ampiezza, riformulare il problema del minimo numero di mosse per far convergere i cavalli in una posizione, sicuramente un vertice raggiungibile da tutte le posizioni iniziali dei cavalli, come un problema di ricerca del cammino minimo. In particolare stiamo parlando di un problema di ricerca del cammino minimo a partire da k sorgenti, corrispondenti alle posizioni iniziali dei cavalli.

Prima di procedere ad analizzare l'implementazione è bene sottolineare alcune considerazioni progettuali fatte sulle strutture dati di tipo PriorityQueue utilizzate. Malgrado sia ampiamente dimostrato, ed in sede di analisi del costo computazionale sarà nuovamente evidente, che la migliore implementazione possibile della coda con priorità sia quella basata su heap di Fibonacci, è stato scelto di usare degli heap binari. Questo perché, data la peculiarità del problema, introdurre una struttura il cui comportamento lazy non poteva essere giustificato da un grande numero di operazioni sui nodi per vertice (mediamente sono 8 operazioni e sono rari i casi discostanti). Quindi il grafo presenta una densità uniformemente distribuita, non insolitamente alta, di archi. Questi archi, oltretutto, sono di costo unitario proprio per la struttura del problema e della lettura dello stesso (almeno per quanto riguarda la parte di creazione del grafo delle mosse): non è quindi ulteriormente giustificato mantenere una coda basata su una priorità proporzionale alla profondità della visita, considerando anche il non indifferente overhead che si viene ad instaurare e che sarà evidente in fase di test.

In generale, per sfruttare l'algoritmo di Dijkstra ci basiamo su un presupposto fondamentale nella ricerca dei

cammini minimi: il passo di rilassamento. Nella ricerca infatti di un cammino minimo a partire da una certa sorgente s, andremo ad aggiornare una frontiera di tutti gli archi uscenti dai vertici raggiunti (banalmente nella prima iterazione sono proprio gli archi uscenti dalla sorgente), sfruttando il seguente passo:

"Sia un grafo orientato G = (V, E) con V insieme dei vertici ed E insieme degli archi, sia T l'albero dei cammini minimi, che include tutti i vertici raggiungibili dal vertice s, scegli un arco (u,v) con $u \in V(T)$ *e* $v \notin V(T)$ *che minimizza la relazione fra le distanze come segue:*

$$D_{sv} \leftarrow D_{su} + w(u,v)$$

dove w(u,v) è la funzione che determina il peso dell'arco (u,v). Effettuato tale passo di rilassamento, aggiungere l'arco all'albero T."

Notiamo che questo appena descritto è un approccio greedy che non permette di riconsiderare gli archi scartati. Notiamo inoltre che è fondamentale implementare la scelta dell'arco durante il passo di rilassamento: per evitare di dover scansionare, nel caso peggiore in $O(m \cdot n)$, tutte gli archi ad ogni vertice, possiamo decidere di mantenere gli archi incidenti all'albero dei cammini minimi T in una coda con priorità costituente proprio la frangia dell'albero. Nel nostro caso, potendo sfruttare le proprietà strutturali dei binary heap che permettono di accedere alla memoria in tempo costante, occupando uno spazio proporzionale al numero dei nodi inseriti, si è scelto di implementare la coda con priorità proprio con tale struttura dati (per ulteriori informazioni consultare l'appendice).

Ma passiamo ad analizzare il codice utilizzato, sempre lasciando i dettagli implementativi delle strutture dati alle appendici di questa relazione:

```
1. def Dijkstra(self, node):
2.       INF = float('inf')
3.       self.validateNodes(INF)
4.       node.set_token(0.0)
5.
6.       T_root = Leaf(node)
7.       T_root.setDistance(node.get_token())
8.       T = Tree(T_root)
9.
10.      leaves = dict()
11.      leaves[node] = T_root
12.
13.      PQ = PriorityQueue()
14.      PQ.insert(T_root, node.get_token())
15.
16.      while not PQ.isEmpty():
17.            u = PQ.deleteMin()
18.            n = u.getElem()
19.            for v in self._adjacency[n.get_index()].getLastAddedList():
20.                if v.get_token() == INF:
21.                     l = Leaf(v)
22.                     leaves[v] = l
23.                     PQ.insert(l, n.get_token() + 1.0)
24.                     v.set_token(n.get_token() + 1.0)
25.                     T.insertLeaf(l, u)
26.                elif n.get_token() + 1.0 < v.get_token():
27.                     relaxed = n.get_token() + 1.0
28.                     leaves[v].setDistance(relaxed)
```

```
29.                     leaves[v].setFather(u)
30.                     leaves[n].addSon(leaves[v])
31.                     PQ.decreaseKey(leaves[v], relaxed)
32.                     v.set_token(relaxed)
33.     return T
```

In questo caso, salviamo nel nodo la sua distanza con lo stesso procedimento con cui si era deciso di conservare l'informazione sull'esplorazione durante la visita in ampiezza. La distanza del nodo è la stessa informazione che viene salvata nella foglia dell'albero (sempre per la semplicità implementativa di cui si era già discusso), ed è usata anche come chiave nella coda con priorità. Notiamo altresì che viene mantenuto un dizionario delle foglie per permettere di aggiornare l'albero T in tempo costante durante la fase di rilassamento, senza aumentare significativamente l'overhead delle operazioni sull'albero in via di formazione.

Come già discusso, è importante che il passo di rilassamento venga compiuto con il minor numero di aggiornamenti della frangia possibile, cercando idealmente di non effettuare O(m) inserimenti nella coda con priorità, ma O(n) (si noti ancora una volta come nel nostro caso questi due valore sono fra loro legati da un rapporto di proporzionalità non particolarmente elevato che di fatto rende già poco “interessante” l'utilizzo dell'algoritmo di Dijkstra). Quindi conservando nella coda con priorità i nodi incidenti all'albero, sarà necessario lavorare sul numero di aggiornamenti delle chiavi della frangia.

Valutando computazionalmente l'algoritmo, assumendo come al solito costanti accessi in memoria ed inizializzazioni, per un solo vertice avremo:

$$T_{Dijkstra}(n) = T_{validation}(n) + T_{priorityQueue}(n) = O(n) + n \cdot T_{deleteMin}(n) + m \cdot T_{decreaseKey}(n)$$

considerando un d-heap binario, quindi per d = 2, possiamo concludere che:

$$T_{Dijkstra}(n) = O(n) + O(n \cdot d \cdot \log_d n + m \cdot \log_d n) = O(m \cdot \log n)$$ [16]

quindi un'esecuzione completa su k vertici, a partire da ognuno dei vertici iniziali occupati dai cavalli, valutando anche la funzione di recupero dei risultati proporzionale al numero dei nodi:

$$T_{visitNodesDijkstra}(n) = T_{makeGraph}(n) + k \cdot T_{Dijkstra}(n) + k \cdot O(n) = k \cdot O(m + n) + k \cdot O(m \cdot \log n) + k \cdot O(n)$$

ottenendo infine:

$$T_{visitNodesDijkstra}(n) = k \cdot O(m \cdot \log n + n)$$

La conclusione è che, come già appurato, l'algoritmo di ricerca del cammino minimo dipende dal numero dei cavalli presenti nell'istanza iniziale. Mentre il tempo di esecuzione dipende, in questo caso, dalla densità del grafo, risultando dall'analisi un tempo computazionale asintoticamente più basso di un fattore $O(m \cdot \log n)$ tanto più denso è il grafo di partenza.

L'Orlo della Fondazione: algoritmo di Floyd-Warshall

Da una considerazione già fatta, potrebbe risultare in alcuni casi conveniente riformulare il problema dei cammini minimi considerando i cammini tra tutte le sorgenti. Una facile implementazione, seppur come dimostreremo particolarmente dispendiosa computazionalmente, è quella di Floyd-Warshall. Durante la fase di

16 E' doveroso notare che se fosse stata necessaria l'implementazione tramite heap di Fibonacci, avremmo $O(m + n \cdot \log n)$

profiling del progetto, questa tecnica algoritmica basata su programmazione dinamica è stata spesso utilizzata come oracolo per gli altri algoritmi già descritti.

Viene quindi qui riportata sia per l'interesse informatico verso questo particolare approccio di programmazione dinamica per la ricerca di cammini minimi, sia per “ragioni storiche” dello sviluppo del progetto stesso.

```
1. def FloydWarshall(self):
2.         INF = float('inf')
3.         nodes, adjacency = self.getNodes()
4.
5.         indexes = nodes.keys()
6.         dim = len(indexes)
7.
8.         dist = [[INF for m in range(dim)] for n in range(dim)]
9.         for i in range(dim):
10.             ind = indexes[i]
11.             dist[ind][ind] = 0.0
12.             adj_nodes = adjacency[ind].getLastAddedList()
13.             for adj in adj_nodes:
14.                 to_ind = adj.get_index()
15.                 dist[ind][to_ind] = 1.0
16.
17.         for k in range(dim):
18.             for i in range(dim):
19.                 for j in range(dim):
20.                     if dist[i][k] != INF and dist[k][j] != INF and dist[i][k] +
            dist[k][j] < dist[i][j]:
21.                         dist[i][j] = dist[i][k] + dist[k][j]
22.         return dist
```

L'algoritmo costruisce prima di tutto una matrice di grandezza calcolabile per $S(n) = O(n^2)$ a cui si accederà con gli indici rispettivi dei nodi del grafo seguendo la struttura delle liste di adiacenza.

Quindi procede seguendo la definizione di cammino minimo k-vincolato, ricordando che nel nostro caso il peso di un arco è unitario, e che la funzione peso è del tipo w(u,v): *“sia un grafo orientato G = (V, E) con V insieme dei vertici ed E insieme degli archi, si definisce un cammino minimo k-vincolato se tutti i sottocammini in esso presenti sono cammini minimi e se vale che:*

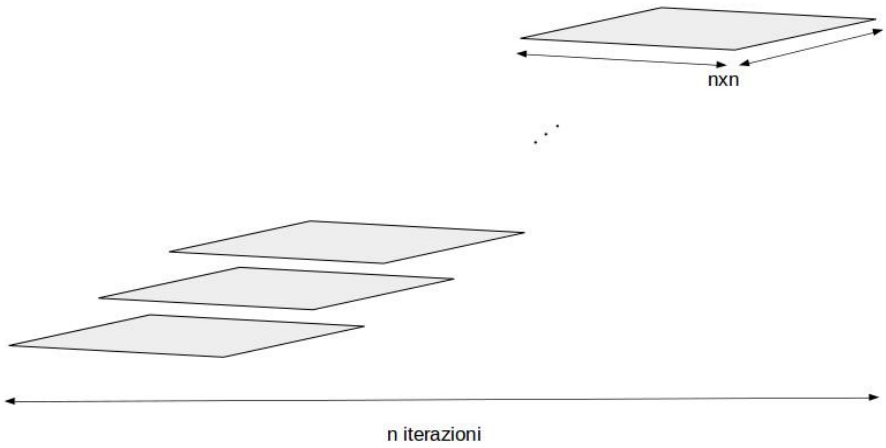
$$d_{xy}^{k} = \begin{cases} w(x,y) \ \ se \ \ (x,y) \in E \\ 0 \ \ se \ \ x = y \\ \infty \ \ altrimenti \end{cases}$$

valendo inoltre che: $d_{xy}^{n} = d_{xy}$ ”

Quindi l'algoritmo itera per la dimensione in nodi del problema sulla matrice, sovrascrivendo ad ogni passo, se necessario, i valori della matrice secondo una semplice regola d'avanzamento, che deriva di fatto sia dalla natura stessa della matrice come codifica delle liste di adiacenza, sia dalla natura stessa dei cammini minimi k-vincolati. Chiamando quindi x, y due generici vertici e v un qualsiasi nodo che permetta di definire un

cammino tra i due, sia inoltre ogni cella della matrice definita qui come la distanza tra i due vertici:

$$d_{xy}^{k} = min\{d_{xy}^{k-1},\ d_{xv_k}^{k-1} + d_{v_k y}^{k-1}\}$$



Da un punto di vista computazionale è facile dimostrare il tempo di ordine cubico d'esecuzione, supposti come al solito costanti eventuali accessi in memoria e confronti

$$T_{minimumFW}(n) = T_{makeGraph}(n) + T_{matrixInitialization}(n) + T_{FW}(n) + T_{readMatrix}(n)$$

poiché la lettura della matrice avviene proporzionalmente al numero dei cavalli ed al più di una lunghezza pari ad un lato della matrice (scansionando la colonna relativa alla posizione iniziale del cavallo) che è proporzionale a sua volte alla grandezza dell'istanza del problema rispetto ai suoi nodi, otteniamo:

$$T_{minimumFW}(n) = k \cdot O(m + n) + O(n^2) + O(n^3) + k \cdot O(n) = O(n^3)$$

Il che rende l'algoritmo eccessivamente lento per una qualsiasi implementazione realistica del problema. Unica eccezione potrebbe essere sollevata sfruttando magari la scacchiera di input come matrice iniziale del problema e lavorare quindi su una struttura già data evitando la costruzione del grafo. Questa implementazione è stata scartata in fase di progettazione per la volontà di risolvere il problema del “salto del cavallo” e fornire una sua astrazione a livello di grafo. Grafo a cui si sarebbe potuti approcciare con diverse soluzioni, implementando diverse procedure e, in ultima analisi, utilizzarlo per risolvere un problema non strettamente specifico alle consegne del progetto stesso.

L'altra faccia della Spirale: il grafo delle mosse come visita in ampiezza

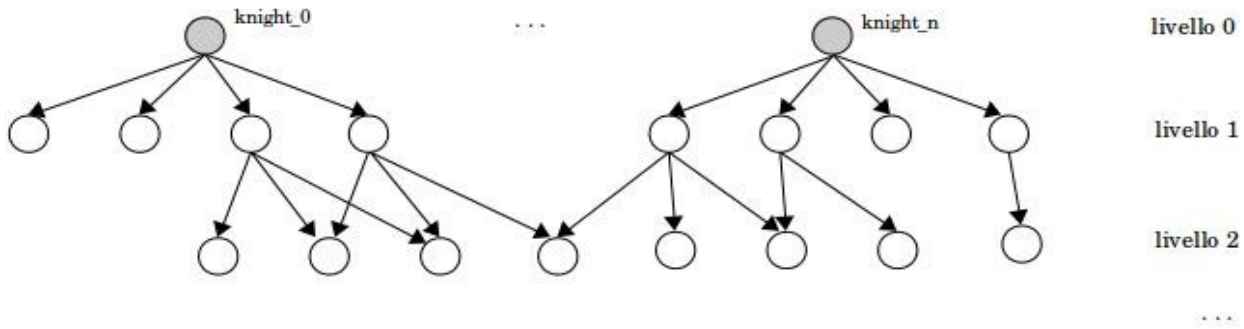
Ricapitolando brevemente a titolo di premessa: le soluzioni viste nell'ultimo capitolo ci hanno permesso di stare “sulle spalle dei giganti”, sfruttando tecniche algoritmiche particolarmente collaudate e note. In particolare dopo aver costruito un grafo delle possibili mosse è stato possibile ragionare su di esso sfruttando strutture dati prevalentemente note per analizzare i possibili cammini minimi da un certo numero di sorgenti a tutte le raggiungibili posizioni.

L'intuizione di una alternativa nasce proprio dal nome scherzoso dato ad ognuna delle sezioni finora viste: la nostra Prima Fondazione è situata, come la sua corrispettiva letteraria, agli estremi di una spirale, di un calcolo del grafo delle mosse, in parte dispendioso, ma che permette una agevole risoluzione tramite una visita in

ampiezza. Ma dove si trova l'estremo di una spirale se non nel suo centro?

Cerchiamo quindi di rileggere sotto una nuova luce l'intero problema, a partire dalla stessa inizializzazione del grafo. A causa della natura stessa del calcolo delle mosse, mantenendo due contatori è possibile costruire una lista di tuple, questa volta così fatte *(position_from, position_to, level).* Stiamo infatti considerando il fatto che ad ogni estensione delle mosse, stiamo generando le mosse il più vicino possibili alla posizione iniziale del cavallo stesso ed è conservabile in tempo costante l'informazione sul livello raggiunto di volta in volta.

Cercando di visualizzare il problema graficamente...



A questo punto basterà riscrivere la funzione makeGraphBFS() come segue, tenendo opportunamente conto delle distanze e di fatto codificando quella che è una visita in ampiezza.

```
1. def makeGraphBFS(self):
2.         r = self.get_rows()
3.         c = self.get_cols()
4.
5.         for k in self._knights:
6.             kgt = self.setGraph().insertNode(k.get_position(), k)
7.             self._knights_nodes.append(kgt)
8.             kgt.set_distance(0)
9.             k.completeTour(r, c)
10.
11.         how_many = len(self.getKnights())
12.         minimum = float('inf')
13.         for knight in self._knights:
14.             for step in knight.getMoves():
15.                 move_from = step[0]
16.                 move_to = step[1]
17.                 node = self.setGraph().insertNode(move_from)
18.                 moveNode = self.setGraph().insertNode(move_to)
19.                 moveNode.set_distance(step[2])
20.             knight.refreshBuffer()
21.
22.         for node in self.setGraph().getNodes()[0].itervalues():
23.             if node.get_count() == how_many:
24.                 if node.get_distance() < minimum:
25.                     minimum = node.get_distance()
26.         return minimum
```

Da un punto di vista dell'analisi computazionale utilizzando le assunzioni fatte precedentemente e assumendo lineare la scansione dei nodi del grafo avremo che:

$$T_{makeGraphBFS}(n) = k \cdot T_{completeTour}(n) + T_{nodes}(n) = k \cdot O(n) + O(n) = k \cdot O(n)$$

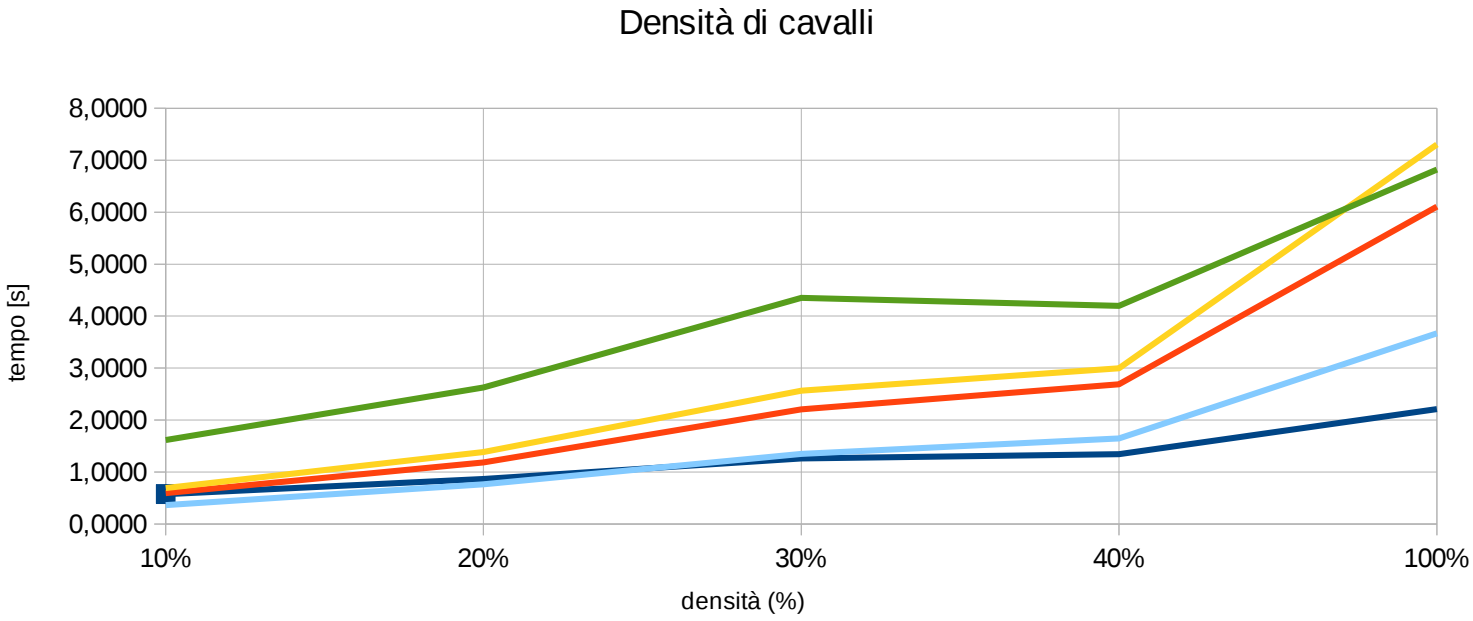
con una occupazione di memoria pari proprio a $S(n) = O(n)$

Otteniamo quindi nuovamente un tempo lineare rispetto al numero dei nodi!

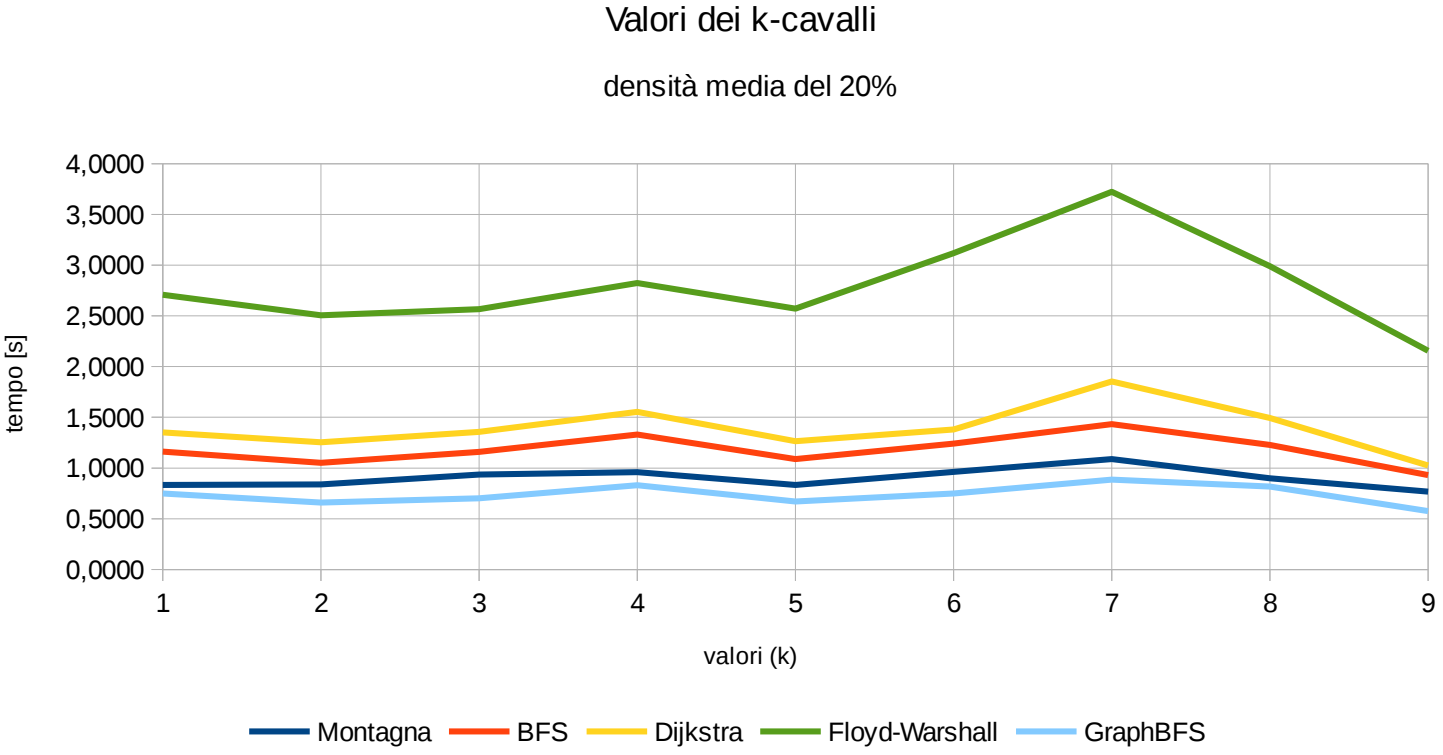
Sfruttando quindi le proprietà del grafo delle mosse rispetto alla sua generazione, sfruttando i dettagli implementativi dei nodi del grafo, eliminando la sovrabbondante codifica degli archi in favore di una più snella analisi delle profondità delle mosse generate, siamo riusciti ad ottenere un algoritmo lineare al numero dei nodi, malgrado il collo di bottiglia del calcolo di tutte le possibili mosse. La relativa semplicità implementativa di questa soluzione la rende una candidata ideale alla risoluzione del nostro problema, anche in vista di un'applicazione pratica.

3. Una questione di proporzioni

Giunti a questo punto abbiamo una collezione di cinque algoritmi. Li andiamo a testare su differenti tipi di istanze[17]. Abbiamo nel primo test modificato la densità della scacchiera, cioè il numero di pezzi presenti a parità di dimensione, mentre nel secondo, mantenendo costante tale densità[18] ad un valore medio, si è fatto variare il numero di cavalli dell'istanza del problema.[19]



17 Si nota che per rispettare l'applicazione effettiva si è scelto di slegare i casi di test dalle dimensioni della scacchiera e di far variare le condizioni con cui muovono i cavalli.
18 E' stata scelta la densità media utilizzata durante la fase di progettazione e di ingegnerizzazione degli algoritmi.
19 Altri casi di test eseguiti che possono essere interessanti per il lettore sono riportati nell'Appendice C.



Il comportamento evidenziato dai grafici dei tempi[20] rispecchia l'analisi computazionale eseguita precedentemente.

Montagna

C'è da fare un'importante premessa prima di analizzare i due casi. Il seguente algoritmo pecca di affidabilità, non è in grado di assicurare al 100% la correttezza della soluzione. Ce ne siamo accorti solamente in questa fase finale di test e abbiamo voluto riportarla comunque per la sua particolare implementazione che permette senza strutture dati esplicite e con una stima probabilistica di risolvere il problema in un buon tempo. Abbiamo valutato che l'algoritmo risponde correttamente il 98% delle volte, andando cioè a sbagliare in media 2 volte su 100. Avremmo voluto correggere questo errore prima della consegna, ma dopo molti tentativi non siamo riusciti a trovare il giusto compromesso. Abbiamo tentato calcolando il target con altre medie pesate. Abbiamo poi aggiunto ai target tutte le posizioni dei cavalli dopo aver osservato che la maggior parte delle volte la soluzione era tener fermo un cavallo e muovere tutti gli altri su questo. Ma non siamo mai riusciti ad avere una correttezza pari al 100% fino a quando non abbiamo aggiunto tutta la scacchiera alla lista targets. Ma quello che volevamo fare era evitare proprio questo! Quindi i test sono stati fatti utilizzando l'algoritmo che in media sbaglia 2 volte su 100, poiché crediamo possa essere interessante e crediamo di essere sulla buona strada. Non siamo stati in grado di dare una stima migliore della posizione del target ma siamo convinti che questa, una volta determinata, possa rendere quest'algoritmo una efficiente e semplice soluzione alternativa al problema.

1. al variare della densità

Questo tipo di test mostra come l'algoritmo dipenda anche dal numero dei cavalli presenti sulla scacchiera. Non è solamente una BFS a partire dal target (una o più a seconda della lunghezza della lista targets) poiché vi è anche il calcolo del target che impiega una quantità di tempo proporzionale al numero dei cavalli e quindi la

20 E' possibile replicare i risultati scegliendo dal dataset presente nel CD-ROM del progetto i rispettivi test.

densità va incidere non solo sul tempo impiegato per trovarli tutti ma anche sul tempo per calcolare il target stesso. Segue però in linea di massima l'andamento di tutti gli altri algoritmi mostrando una velocità maggiore non andando ad implementare strutture dati quale quella del grafo.

2. al variare dei valori dei cavalli

Questa volta vediamo che per tutti i casi il tempo di esecuzione si aggira attorno il secondo, cosa che accadeva già nel test precedente con densità del 20%. Le variazioni sono dovute ai casi particolari infatti si può vedere che segue l'andamento degli altri algoritmi. Il variare del k non inficia in alcun modo sul tempo computazionale.

Grafo delle mosse

1. al variare della densità

In questo caso prendiamo come caso migliore la visita in ampiezza sul grafo delle mosse. Notiamo infatti che la variante con visita in ampiezza durante la costruzione del grafo ha un costo computazionale davvero poco importante, a fronte di una semplicità implementativa di cui si è già discusso. Notiamo inoltre l'andamento similare con la visita in ampiezza su un grafo già formato, che comincia a discostarsi solamente al tendere dalla saturazione massima del grafo: in questa situazione è infatti particolarmente pronunciato il numero di archi, che sarà nel caso peggiore proprio pari al numero massimo di mosse (8 per una scacchiera di dimensioni non trascurabili). In questo caso, appunto, l'overhead della visita in ampiezza effettuata da numerosi punti del grafo, viene compensato durante la costruzione del grafo delle mosse: tale compensazione è visibile nella lieve diminuzione della pendenza della curva.[21]

Prendendo quindi a metro di paragone la visita in ampiezza, particolarmente importante diventa lo studio del caso dell'algoritmo di Dijkstra. Viene qui infatti giustificata la sua esclusione da una applicazione pratica: poiché il grafo delle mosse presenta tutti archi di costo unitario, nella coda con priorità avremo un aumento significativo dell'overhead nel mantenimento della frangia di un numero di archi di peso uguale. Questo porta a inconcludenti chiamate alla funzione di ripristino della proprietà di ordinamento, che potrebbero chiaramente essere sostituite da una generica coda! Particolarmente importante è notare come tale fenomeno possa essere così discriminante, addirittura nei confronti dell'algoritmo di Floyd-Warshall: il mantenimento della coda con priorità, su una ricerca per ognuno dei cavalli, diventa pesante e poco gestibile durante i generici passi di rilassamento.

Infine Floyd-Warshall fornisce un interessante caso di profiling[22] . Malgrado il tempo di esecuzione cubico nel numero dei nodi, poiché la dimensione del problema rispetto al numero dei nodi è meno variabile rispetto al numero degli archi, quindi delle possibili mosse calcolabili, dopo un iniziale (e confrontabile con gli altri casi) aumento del costo computazionale, la variazione dello stesso risulta inferiore rispetto agli altri. In particolare su densità molto elevate, il disaccoppiamento dal numero di archi delle dimensioni rilevanti per il tempo

21 La scelta accorta dei valori di densità risulta qua particolarmente vantaggiosa: la costruzione del grafo delle mosse risulta tanto più variabile tanto è più basso il numero dei cavalli, poiché ogni cavallo continuerà ad inserire nuovi nodi invece che procedere alla sola fase di collegamento degli stessi.

22 E' importante sottolineare che tutti i casi di test in questa sessione sono stati presi su una attenta media dei risultati, tenendo conto anche delle condizioni della macchina su cui girava il programma, su quanto tempo fosse già accesa e su quanto questo avesse influito sul carico della cache.

d'esecuzione fornisce un aumento delle prestazioni in prossimità della saturazione del grafo. Tuttavia questa caratteristica non è sufficiente per considerare una applicazione reale di quest'algoritmo in vista della scelta della soluzione migliore: nel caso medio risulta eccessivo il costo computazionale, malgrado la semplicità implementativa che si è già avuto modo di constatare.

2. al variare dei valori dei cavalli

Basandoci sugli studi e sul profiling di cui si è già discusso, questo caso, basato su una densità di cavalli media del 20%, riesce a rendere particolarmente bene le differenze a parità di costruzione del grafo delle mosse. E' infatti immediato constatare la maggior efficienza dell'algoritmo che si basa sull'implementazione di una visita in ampiezza durante la fase di creazione del grafo e le verifiche dell'analisi computazionale, una volta resi poco influenti i parametri relativi al numero dei cavalli, che riportano il costo lineare rispetto al numero dei nodi.

Notiamo che l'andamento delle curve presenta un'unico punto di discostamento delle variazioni, a meno delle traslazioni dovute alle cause già descritte, nel caso in cui i cavalli raggiungono un numero di mosse per turno pari a 7, si è infatti verificata, secondo le nostre supposizioni, in quella occasione, un interessante aumento della densità locale dovuta alla componente randomica del generatore di input. In tal caso Dijkstra subisce un pericoloso picco, dovuto alla transizione non ben gestita dalla proprietà di ordinamento della coda con priorità. Mentre Floyd-Warshall trova un numero di nodi abbastanza elevato rispetto al numero di archi, ma riesce ugualmente a conservare la sua peculiare variazione mediamente lineare nei tempi d'esecuzione.

In generale l'andamento tende ad essere abbastanza limitato nelle sue variazioni: ciò è dovuto essenzialmente alla dipende dal numero di nodi[23] e di archi, che rimane invariato per tutte le istanze. Le oscillazioni sono quindi imputabili alla codifica stessa delgi algoritmi ed all'analisi delle mosse, come già discusso.

4. Scacco matto!

note e considerazioni conclusive

I risultati ottenuti sono frutti di alcuni mesi di progettazione e di interpretazione del problema, piuttosto che di programmazione. Sia la visione del problema come una visita a partire da un ipotetico centro della disposizione dei cavalli, sia la strutturazione del problema come grafo delle mosse, hanno richiesto qualche sforzo per consentire sia di avere una codifica modulare, come più è volte è stato sottolineato, sia per essere chiaramente coerenti da consentire i vari approcci descritti.

Le conclusioni che possiamo trarre dopo questa trattazione sono particolarmente favorevoli rispetto al grafo delle possibili mosse. Infatti a parità di mosse calcolabili sulla scacchiera, la costruzione di tale grafo si è rivelata inizialmente uno strumento formidabile per permettere di sfruttare le nostre conoscenze sullo studio dei grafi, tanto quanto in ultima analisi una codifica della visita su tutta la scacchiera che si è più volte cercato di ottenere.

23 La dipendenza dai nodi non è stata considerata particolarmente rilevante durante i casi di test perché si è preferito concentrare l'attenzione sulla dipendenza dai valori e dal numero dei cavalli. Abbiamo cercato infatti di immaginare una applicazione reale dell'algoritmo, dove le dimensioni della scacchiera sarebbero più un limite fisico al calcolo delle mosse che ad una risoluzione della ricerca di cammini minimi, aspetto su cui ci si è invece soffermati.

La soluzione scelta ricade, quindi, sulla costruzione progressiva delle mosse di ogni cavallo e sul mantenimento quindi di una struttura di nodi che permetta di aggiornare, ad ogni iterazione sull'insieme dei cavalli, la distanza di ognuno dei nodi da ognuno dei cavalli, a seconda delle possibili mosse effettuate da un cavallo in un turno.

Tuttavia abbiamo mantenuto tutte le implementazioni che ci hanno permesso di codificare una soluzione possibile. Questo per la nostra volontà di mantenere traccia del lavoro svolto per ulteriori applicazioni e sviluppi.

Roma, 28/08/2014

Appendice A: strutture dati

Prima di parlare delle strutture dati utilizzate per implementare le soluzioni di cui si è parlato, è necessario fare una piccola premessa sulla notazione utilizzata.

- Nel caso di strutture dati che costituiscono delle “primitive”, come le liste, le foglie dell'albero, l'albero stesso, nodi e struttura dell'heap, attributi e metodi sono stati implementati con una formattazione *Camel Case*. Stesso discorso vale per i metodi delle classi che implementata la struttura del grafo e del pezzo del cavallo, nel caso in cui tali metodi siano una parte attiva dell'algoritmo o forniscano una interfaccia per ulteriori strutture;
- Nel caso di attributi e metodi utilizzati per le classi che implementano il match, il grafo ed i nodi od il cavallo, nel caso questi non facciano parte attiva dell'algoritmo, è stata utilizzata una formattazione rispettando lo standard *PEP-8*.

E' altrettanto importante far notare che molte delle interfacce codificate nelle classi che verranno esposte non sono state effettivamente utilizzate, ma si è scelto di mantenerne traccia per permettere al codice di essere riusabile per futuri studi ed implementazioni.

Nota: per la codifica di lista doppiamente collegata, coda, albero e coda con priorità si è tratto spunto dalle strutture dati viste ad esercitazione, con opportune modifiche per ottimizzare il tempo computazionale o per fornire un'adeguata interfaccia rispetto alla struttura del problema.

Match

La classe match astrae l'intera partita andando a conservare le caratteristiche della scacchiera e la lista di tutti i cavalli presenti su essa. Nel caso dell'algoritmo della montagna conserva anche il numero dei cavalli trovati e il numero dei turni necessari per finire la partita. Vediamo ora alcuni metodi utilizzati precedentemente nell'algoritmo montagna:

get_knights() $O(1)$

restituisce l'intera lista dei cavalli presenti nel match.

is_finished() $O(1)$

restituisce True nel caso il match sia stato completato, cioè se tutti i cavalli sono stati trovati dalla visita a partire dal fake knight, False altrimenti.

knight_found(knight, distance) $O(1)$

aumenta il contatore che mantiene il numero dei cavalli trovati. Richiama inoltre i metodi che permettono di calcolare i turni necessari dato la distanza e il k del cavallo. Non restituisce nulla.

get_turns() $O(1)$

restituisce il numero dei turni necessari per completare il match

finish(force) $O(1)$

tenta di chiudere il match nel caso questo non sia possibile solleva una eccezione. Nel caso sia stato passato il valore booleano force settato a True forza la chiusura del match senza sollevare eccezioni.

reset() $O(k)$

resetta il match andando a riportare allo stato originale tutti gli attributi che mantengono i cavalli e che mantiene il match stesso.

makeGraph() $O(k \cdot m + k \cdot n)$

permette di costruire da k cavalli iniziali un intero grafo delle mosse procedendo come segue: vengono inizializzati i nodi contenenti i cavalli nelle proprie posizioni iniziali, quindi vengono calcolati per ognuno dei cavalli i tour complete. Per ognuna delle mosse del tour viene effettuato un inserimento e vengono collegate le mosse nelle liste di adiacenza dei nodi corrispondenti, nel caso sia necessaria inserire un nodo non ancora presente nel grafo, altrimenti al nodo viene solamente aggiornata la lista di adiacenza.

makeGraphBFS() $O(k \cdot n)$

permette di attuare una visita completa di tutte i nodi corrispondenti alle possibili mosse dei cavalli, salvando durante la fase di creazione la distanza da ogni cavallo. Una semplice scansione dei nodi permette di ritornare il minimo numero di mosse, oppure infinito nel caso non sia possibile raggiungere una casella comune.

minMovesBFS() $O(k \cdot m + k \cdot n)$

permette di attuare una visita in ampiezza sul grafo delle mosse opportunamente costruito, a partire dai nodi indicanti le posizioni iniziali dei cavalli. Una semplice scansione dei nodi permette di ritornare il minimo numero di mosse per raggiungere una casa della scacchiera, oppure infinito nel caso non sia possibile raggiungere una casella comune.

minMovesDijkstra() $O(k \cdot m \cdot \log n + k \cdot n)$

permette di attuare una ricerca dei cammini minimi tramite l'algoritmo di Dijkstra, a partire dai nodi indicanti le posizioni iniziali dei cavalli. Una semplice scansione dei nodi permette di ritornare il minimo numero di mosse per raggiungere una casa della scacchiera, oppure infinito nel caso non sia possibile raggiungere una casella comune.

minMovesFloydWarshall(dist) $O(n^3)$

permette di attuare una ricerca dei cammini minimi tra tutti i nodi del grafo sfruttando la programmazione dinamica. Quindi una scansione delle colonne della matrice corrispondenti ai nodi contenenti i cavalli nelle posizioni iniziali permette di ritornare il minimo numero di mosse per raggiungere una casa della scacchiera, oppure infinito nel caso non sia possibile raggiungere una casella comune.

Knight

Rappresenta il cavallo e principalmente mantiene la sua posizione e calcola le mosse.

I suoi metodi principali:

get_moves(row, col) $O(1)$

restituisce calcolando le possibili mosse del cavallo nella scacchiera di dimensioni row e col.

get_other_moves(row, col) $O(1)$

restituisce calcolando le possibili mosse del k-cavallo nella scacchiera di dimensioni row e col aumentando di uno il k. Questo vuol dire che non verranno calcolate le posizioni convenzionali raggiungibili da un convenzionale cavallo ma le posizioni successive come spiegato nell'introduzione.

calculateWeight(dist) $O(1)$

permette di calcolare, a seconda del valore k proprio del k-cavallo, a partire dalla distanza dist specificata, il numero di mosse necessarie al cavallo per coprire tale distanza.

singleMove(pos, rows, cols) $O(1)$

permette di calcolare le mosse possibili, rispetto ad una posizione del cavallo e rispetto alle dimensioni della scacchiera. Mantiene in un buffer le mosse precedentemente calcolate per evitare di inserire duplicati.

completeTour(rows, cols) $O(n)$

permette di calcolare un intero tour del cavallo, evitando di incorrere in cammini circolari e salvando per ognuna delle mosse una tupla (position_from, position_to, level) di modo da permettere la costruzione del grafo delle mosse e di poter inizializzare ogni nodo con la distanza dalla posizione iniziale del cavallo.

Lista doppiamente collegata

Questa struttura dati, codificata come DoubledLinkedList, è stata utilizzata al posto delle liste built-in di Python per permettere la costruzione di una classe rappresentante il grafo che fosse il più flessibile e riusabile, visto il grande numero di test e di sperimentazioni che si erano previsti in fase progettuale.

E' costituita da una unità elementare definita come Record che consente di mantenere due puntatori, al record precedente ed al record successivo, e di conservare al suo interno una determinata informazione.

Sfruttando il record come unità fondamentale è possibile effettuare inserimenti e cancellazioni in tempo costante *O(1)* e di effettuare una intera scansione in *O(n)*, per n inserimenti effettuati. Varranno inoltre due proprietà generali:

- (forte) è possibile aggiungere o togliere record alla struttura mantenendone la coerenza;
- (debole) l'ordine dei record non richiede di essere consecutivo;

Nella codifica utilizzata in questo progetto si è scelto di mantenere un buffer degli ultimi inserimenti effettuati. Tale decisione è stata presa sulla base della considerazione che, data la natura del grafo, non ci sarebbero mai state cancellazioni di record. Tale buffer è utilizzato sia per il *linking* dei nodi, per evitare inserimenti doppi, sia come ausilio in seguito per la scansione delle liste di adiacenza per aumentare l'efficienza computazionale mantenendo lo spazio utilizzato complessivamente lineare al numero dei nodi.

In particolare, nell'implementazione del grafo sono stati usati:

addAsLast(elem) $O(1)$

permette di aggiungere un record contenente l'elemento elem alla fine della lista doppiamente collegata, modificando e aggiornando opportunamente i puntatori in tempo costante.

getList() $O(n)$

ritorna una lista di tutti gli elementi conservati nei record della struttura dati, necessita di implementare una scansione dell'intera lista.

Tramite la definizione data adesso, è possibile codificare una classe derivata Queue per rappresentare una coda, molto utile ad esempio nel caso di una visita in ampiezza sull'intero grafo, performante in tempo costante *O(1)* sia l'operazione di aggiunta nella coda, sia l'operazione di rimozione dell'elemento in testo alla coda. Opera secondo la logica First In First Out (FIFO).

enqueue(elem) $O(1)$

permette di inserire un elemento nella coda rispetto l'ordine di visita.

dequeue() $O(1)$

ritorna l'elemento in testa alla coda, rispetto l'ordine di inserimento della enqueue.

Grafo

La rappresentazione del grafo, in particolare implementato tramite liste di adiacenza, è stata già discussa vista la sua importanza nel paragrafo dedicato proprio alla creazione del grafo delle mosse. Riportiamo quindi qui brevemente il costo computazionale delle due operazioni più comuni compiute sul grafo, ad eccezione degli algoritmi già approfonditi in apposita sede:

insertNode(elem, weight) $O(1)$

permette di inserire un nuovo nodo nel grafo: nel caso questo nodo sia già presente viene ritornato.

linkNode(tail, head) $O(1)$

permette di collegare due nodi, inserendo nella lista di adiacenza del nodo di coda il nodo di testa, di fatto permettendo una codifica implicita dei rami del grafo.

Nota: per ottenere un tempo costante anche in presenza di verifiche sull'esistenza in memoria di nodi già inizializzati, è stata sfruttata la velocità dell'accesso in memoria garantito dall'implementazione in Python delle sue liste built-in.

Albero

La struttura dati Tree viene ampiamente utilizzata, sia nelle sue unità strutturali, le foglie della classe Leaf, sia per le sue proprietà, con lo scopo di mantenere un albero dei cammini minimi sia nel risultato della visita in ampiezza sia come risultato dell'algoritmo di Dijkstra.

La generica foglia di un albero, definendo come foglia “degenere” anche la radice stessa, cioè nel nostro caso il vertice da cui si è calcolato il cammino minimo, contiene molto semplicemente un puntatore al padre ed una lista di figli. Stiamo quindi codificando l'albero tramite una rappresentazione collegata con una lista figli, sempre per garantire le massime flessibilità e riusabilità possibili in sede di implementazione.

Per quanto riguarda la foglia, codificata nella classe Leaf, dell'albero abbiamo codificato, in tempo costante, le seguenti operazioni:

setFather(leaf) $O(1)$

setta il puntatore della foglia al padre. Inoltre modifica la distanza della foglia dalla radice, settandola ad un valore di una unità più grande rispetto alla distanza del padre stesso.

addSon(leaf) $O(1)$

aggiunge alla lista di figli della foglia un nuovo figlio.

setDistance(dist) $O(1)$

modifica la distanza della foglia della radice settandola ad un valore pari a dist.

Per quanto riguarda l'albero stesso invece, sono stati utilizzati maggiormente i metodi:

insertLeaf(sonLeaf, fatherLeaf): $O(1)$

inserisce nell'albero una nuova foglia, modificando opportunamente il suo puntatore al padre, appartenente già all'albero anche se il corretto uso dell'interfaccia è dato all'utente, e modificando opportunamente la lista di figli del padre. Aggiunge inoltre la nuova foglia alla lista di foglie dell'albero.

getLeaves(): $O(1)$

ritorna in una lista le foglie dell'albero.

Nota: nel nostro caso l'albero può essere al profondo al più *h*, cioè, come già visto, la massima profondità raggiunta dalle iterazioni durante la creazione del grafo delle mosse.

Coda con priorità basata su binary-heap

Come già spiegato nel paragrafo riguardo l'implementazione dell'algoritmo di Dijkstra, si è scelto di

implementare una coda con priorità, cioè che mantiene un heap il cui primo elemento è un elemento che minimizza una funziona di priorità. Nel nostro caso la funzione di priorità è la distanza che lega i vertici incidenti sull'albero dei cammini minimi.

In particolare si è deciso di sfruttare i binary-heap nella coda con priorità, cioè alberi che possono avere per foglia al più due figli. Per consentire tale codifica è stato deciso di seguire lo schema visto ad esercitazione codificare in una classe BinaryHeapNode che conserverà l'elemento con cui è stato inizializzato, una chiave che rappresenta la sua priorità nell'heap, ed un indice.

Proprio sfruttando l'indice, cioè la posizione all'interno dell'heap, è possibile risalire alla posizione di tutti i figli del nodo, senza necessità di mantenere una lista di indici ai figli. Sono infatti vere, per un per un generico d-heap le due relazioni posizionali:

$$heapNode_{son} = HEAP[d \cdot heapNode_{father} + i] \quad per \quad i = 0 \;\; a \;\; d-1$$

$$heapNode_{father} = HEAP[heapNode_{son} \;/\; d]$$

Un attributo che si è deciso di aggiungere per aumentare l'efficienza di alcune operazioni di cui ora si discuterà, è un dizionario che associa ad ognuno degli elementi inseriti nell'heap il suo nodo. Questo per fornire un'interfaccia più comoda all'utente.

Vediamo quindi i metodi utilizzati, tenendo conto che, rappresentato l'heap come lista, ad ogni nuovo inserimento viene aggiunto in fondo alla lista il nuovo nodo e si procede ad aggiornare la lista in basa alla priorità:

moveUp(node) $O(\log n)$

finché il nodo non è pari proprio alla radice e finché la sua chiave è più grande della chiave del padre, scambia con una apposita funzione di swapping il nodo padre con il nodo figlio. Operazione al più costosa tanto quanto l'altezza dell'heap, cioè con costo logaritmico nel nostro caso.

insert(elem, key) $O(\log n)$

aggiunge un nuovo nodo nell'heap binario e chiama una moveUp sullo stesso nodo per ripristinare la proprietà di ordinamento in base alla priorità del nodo.

decreaseKey(node) $O(\log n)$

viene aggiornata la chiave del nodo e viene chiamata una moveUp sullo stesso nodo per ripristinare la proprietà di ordinamento in base alla nuova priorità del nodo.

deleteMin() $O(1)$

viene estratto il nodo con la minima priorità nell'heap: ciò è garantito dalle chiamate alla moveUp durante la fase di inserimento e di aggiornamento delle chiavi.

Appendice B: generatore di input/output

Il generatore di input/output segue le seguenti specifiche:

- Input:
    1. La prima riga del file input.txt contiene un intero T (≤ 100), che indica il numero di casi di test.
    2. Ogni caso di test inizia con una riga vuota.
    3. La riga successiva contiene due interi m, n (1 ≤ m, n ≤ 10) che rappresentano rispettivamente il numero di righe e di colonne della scacchiera.
    4. Ognuna delle m righe successive contiene n caratteri che rappresentano la scacchiera. Numeri che stanno ad indicare il valore k del cavallo o il carattere '.' per indicare una casella vuota.
- Output:
    1. Per ogni caso di test, devi stampare il numero relativo del caso e il risultato desiderato. Se nel caso in esame non è possibile muovere tutti i K-cavalli in una stessa casa della scacchiera, la riga relativa dovrà riportare la parola impossibile.

La parte di codice che genera il file di input è divisa in due moduli. Il primo modulo crea un numero di partite T, il secondo modulo genera un file input.txt vero e proprio. Vediamo il primo modulo:

Il primo modulo è chiamato input_generator.py e contiene due funzioni, generate_chessboard() e generate_input(). La prima va a generare una scacchiera come definito nei punti 3 e 4 delle specifiche. Prende come parametri i valori m ed n e restituisce una lista che rappresenta la scacchiera. Prima di tutto si assicura che il primo sia maggiore uguale di 1 e il secondo minore o uguale di 10, dopodiché appende la prima riga con le informazioni riguardanti la scacchiera e infine crea in modo randomico la scacchiera. Il codice è il seguente:

```
1. def generate_chessboard(m, n):
2.     assert m >= 1, "The number of the rows must be greater then 1"
3.     assert n <= 10, "The number of the columns must be smaller then 10"
4.
5.     chessboard = []
6.     info = [str(m), " ", str(n), "\n"]
7.     chessboard.append(info)
8.
9.     for i in range(0, m):
10.         row = []
11.         for j in range(0, n):
12.             rand = randint(1, 10)
13.
14.             if rand % 7 == 0:
15.                 k = randint(1, 9)
16.                 row.append(str(k))
17.             else:
18.                 row.append(".")
19.         row.append("\n")
20.         chessboard.append(row)
21. return chessboard
```

La parte più interessante del codice è sicuramente il doppio for al cui interno è presente la generazione casuale

dei pezzi da inserire nella scacchiera. Per determinare se si andrà ad inserire un cavallo o meno si è scelto di prendere un numero casuale tra 1 e 10 con la funzione bult-in randint(). Preso questo numero random tra 1 e 10 si applica l'operatore binario % tra il numero e il valore 7. Questo operatore restituisce il resto della divisione tra quei due numeri. Se il risultato è diverso da 0 (e quindi non è un numero divisibile per 7) non sarà un cavallo e la posizione sarà vuota, inseriremo il carattere '.', altrimenti se il risultato è 0 (divisibile quindi per 7) bisogna inserire un cavallo. Si effettua un'altra chiamata alla funzione randint() tra 1 e 9 per la generazione di un numero random che indicherà il k del cavallo. È quindi chiaro che il controllo che permette di stabilire se una casa sarà occupata o meno da un cavallo si basa sul resto per la divisione per 7 del valore generato randomicamente in precedenza. Essendo questo valore generato tra 1 e 10 l'unico per il quale il resto per 7 faccia 0 è solamente 7. Quindi la probabilità che esca un cavallo è 1/10, 10%. Abbiamo fatto questa scelta perché avevamo bisogno di scacchiere non densissime e ci è sembrato un giusto compromesso. Questa scelta è dovuta alla nostra decisione di scegliere se inserire un cavallo o meno controllando se un numero era divisibile o meno per un altro.

La seconda va a generare il file di input come definito dai punti 1 e 2 delle specifiche. Prende come parametro il valore t e restituisce una lista contenente t match. Ci si assicura che il numero dei test sia minore uguale a 100, dopodiché si inserisce una riga contenente il numero che indica quanti match saranno listati, una riga vuota e poi tutti i vari match. Il codice è il seguente:

```
1. def generate_input(t):
2.     assert t <= 100, "The number of the tests must be smaller then 100"
3.
4.     file = []
5.     info = [str(t), "\n"]
6.     file.append(info)
7.
8.     for i in range(0, t):
9.         m = int(10 * random()) + 1
10.        n = randint(1, 10)
11.        chess = generate_chessboard(m, n)
12.        file.append(chess)
13. return file
```

Non c'è molto da commentare in questo caso, quello che si fa è richiamare t volte la funzione generate_chessboard() passando come valori m ed n generati randomicamente. Dovendo essere m maggiore di 1 si è scelto di generarlo usando la funzione bult-in random(), invece per n dovendo essere minore o uguale a 10 si è scelto di usare la funzione già vista prima randint() tra 1 e 10.

Grazie a queste due funzioni contenute in questo modulo siamo in grado di generare tutti gli input che vogliamo e immagazzinarli in una lista. Ancora non è stato creato un file txt come richiesto, per fare questo ci avvaliamo di un'ulteriore modulo, il modulo txt_generator.py. All'interno di questo file sono presenti due funzioni. La funzione generate_file() è il cuore del modulo. Questa funzione riceve come parametri un intero che determina se il file da generare è un file di input (in caso sia 0) o output (1), una lista che contiene le informazioni da salvare sul file, una stringa che rappresenta il percorso dove generare il file e un intero che indica il numero del file.

```
1. def generate_file(kind, text, path, num=None):
2.     global file
3.     if kind == 0:
```

```
        try:
            if num is None:
                name = generate_name_file(0, path)
                file = open(path + name + '.txt', 'w')
            else:
                file = open(path + 'input_' + str(num) + '.txt', 'w')
             for row in text:
                 for col in row:
                     for elem in col:
                         file.writelines(str(elem))
                 file.write("\n")
         finally:
             file.close()
    elif kind == 1:
        try:
            if num is None:
                name = generate_name_file(1, path)
                file = open(path + name + '.txt', 'w')
            else:
                file = open(path + 'output_' + str(num) + '.txt', 'w')
            i = 1
            for elem in text:
                file.write("Caso " + str(i) + ": " + elem + "\n")
                i += 1
        finally:
            file.close()
```

Nulla di particolare anche qui. Il codice verifica se il file da generare è un file di input o output, fa il controllo sull'intero passato e nel caso non sia dato andrà a richiamare la funzione generate_name_file() che genera un nome in ordine sequenziale. Quello che fa consiste nel verificare nella cartella che file sono presenti. Nel caso stessimo generando file di input se è presente il file input_0.txt, input_1.txt genererà il file input_2.txt in modo tale da non sovrascrivere gli altri file di input. Nel caso invece venga passato anche l'intero questo andrà a creare il file di input con quel nome e nel caso già presente lo sovrascriverà. Stesso controllo viene fatto nel caso di file di tipo output.

La funzione generate_name_file() è la seguente:

```
def generate_name_file(kind, path):
    global filename
    i = 0
    while True:
        if kind == 0:
            filename = "input_%d" % i
        elif kind == 1:
            filename = "output_%d" % i
        if not os.path.exists(path + filename + ".txt"):
            return filename
i += 1
```

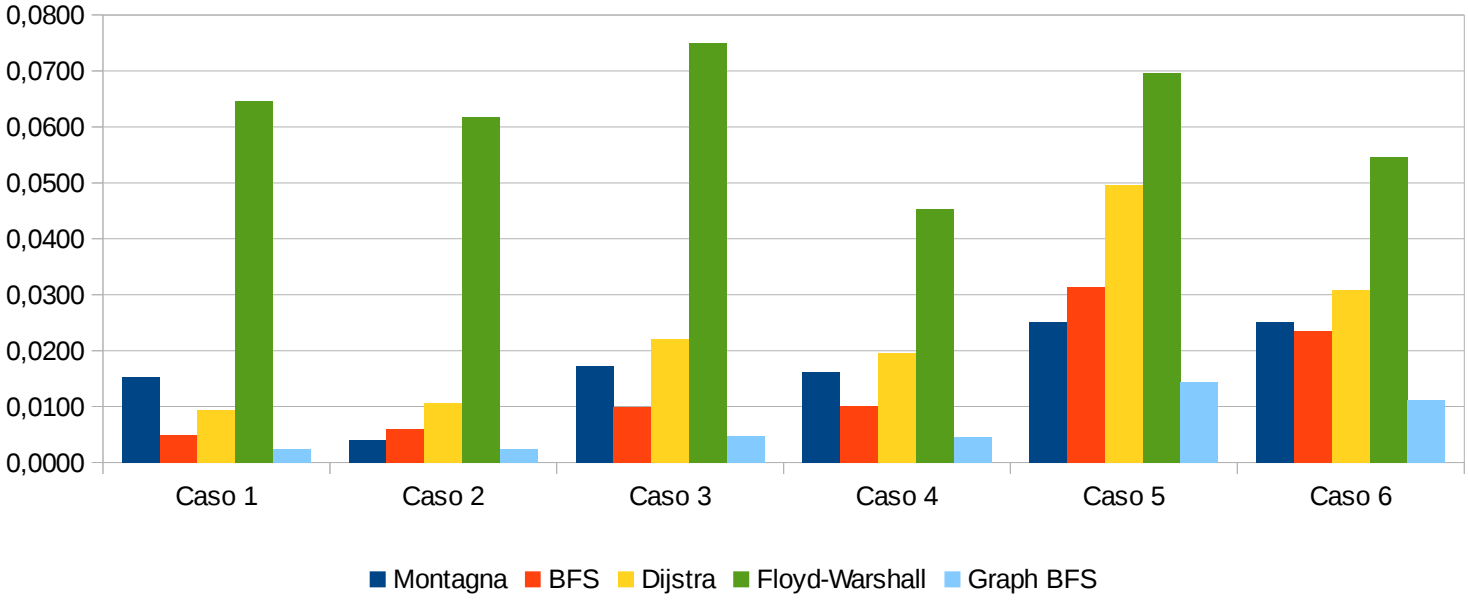
Come detto in precedenza verifica grazie alla funzione bult-in exists se è presente o meno il file andando ad incrementare ogni volta un contatore (il che risulta molto dispendioso in caso di molti file ma per l'uso che ne facciamo va più che bene).

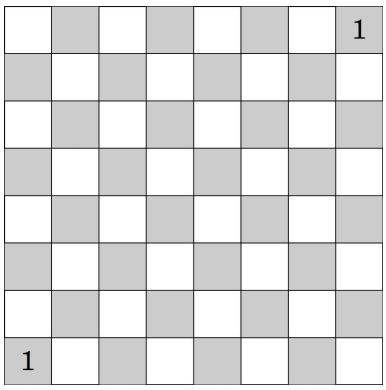
Grazie a questi due moduli e queste poche righe di codice siamo in grado di generare test e creare file di input e

output.

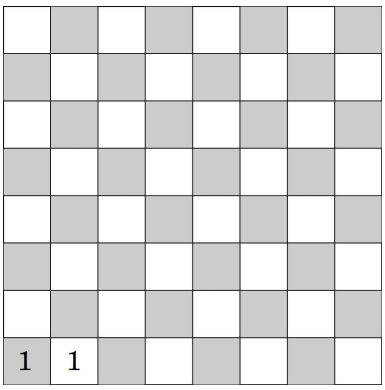
Appendice C: alcuni casi di test interessanti

Analizziamo in questa sede alcuni casi di test interessanti, su istanze del problema che presentano particolari condizioni, sia di disposizione dei pezzi sulla scacchiera, sia per valore dei cavalli. Si è voluto infatti riportare l'andamento degli algoritmi su questi casi minori per poter isolare e valutare l'efficacia sia del calcolo del target, sia della creazione del grafo delle mosse e sua annessa visita. Si noterà come si confermano i risultati di cui già si è discusso precedentemente.

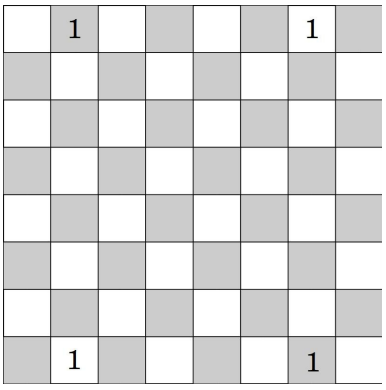




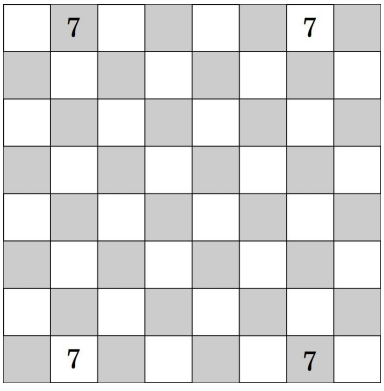
Caso 1



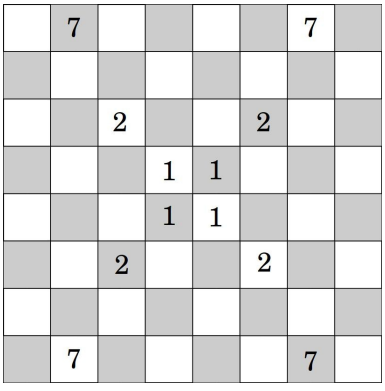
Caso 2



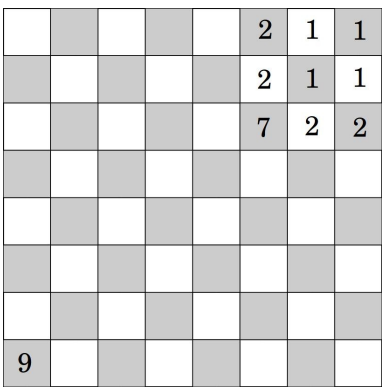
Caso 3



Caso 4



Caso 5



Caso 6

Questi test sono stati compiuti durante la fase di ingegnerizzazione per osservare come i differenti algoritmi si approcciassero al problema.[24]

La soluzione che si è deciso di adottare, ovvero una visita in ampiezza codificata nella generazione del grafo delle mosse, così come la sua controparte meno efficiente, si mantengono entro i comportamenti già descritti, mentre sia Dijksra che Floyd-Warshall stentano a tenere il passo.

Per le visite in ampiezza osserviamo come sia interessante la differenza di costo computazionale apportata dalla codifica implicita di tale visita durante la creazione del grafo delle mosse, osservando come gli algoritmi siano messi “a dura prova”, malgrado rimanendo i migliori, solamente nel caso in cui il grafo risulta sparso ma denso di cavalli oppure fortemente asimmetrico (si notino in particolar modo i casi 5 e 6). In tali circostanze, infatti, è importante nella sua struttura il grafo delle possibili mosse e la sua costruzione richiede visite particolarmente lontane dagli addensamenti principali prima di determinare una possibile posizione finale in comune a tutti i cavalli. Ulteriormente si nota come codificare implicitamente la visita in ampiezza durante la fase di costruzione permette di ottenere in queste circostanze riscontri immediati, malgrado la presenza di zone del grafo dense di archi ma povere di posizioni comuni, ogni qual volta viene raggiunto un nodo, senza la necessità di rielaborare la visita in seconda istanza.

Per l'algoritmo di Dijkstra è facile intuire che valgono le considerazioni di cui si è già trattato, a meno di tener conto di un ulteriore fattore: nel caso in cui il grafo presenta notevoli raggruppamenti di cavalli, il tempo di esecuzione di Dijkstra diventa confrontabile con la visita in ampiezza sul grafo inizializzato con le possibili mosse. Questo a causa del numero di archi incidenti su una posizione già occupato da un cavallo, quindi con la riduzione durante le iterazioni dei nuovi inserimenti ed una, come al solito, limitata rete di archi che li connettono. I passi di rilassamento tendono a diminuire a fronte di una maggior rapidità d'esecuzione.

Notiamo infine che l'algoritmo di Floyd-Warshall presenta picchi notevoli quando il grafo è particolarmente sparso e povero di cavalli dai valori elevati. In tal caso il grande numero di nodi non è compensato da un numero di cavalli sufficiente, quindi vengono effettuati numerosi inserimenti nuovi nel grafo ad ogni iterazione sull'insieme dei cavalli. Allo stesso modo notiamo che il suo comportamento tende a stabilizzarsi, se non addirittura a migliorare, nei casi in cui il grafo presenta una notevole dispersione o una notevole asimmetria: non essendo necessario dipendere dal numero di archi incidenti su ogni nodo durante l'esecuzione dell'algoritmo ma codificando le liste di adiacenza in una matrice dei nodi, può contare sulla rapidità dell'accesso sequenziale alla matrice e sulla regolarità della dimensione della stessa a parità di fattori, per poter mantenere un tempo computazionale sempre maggiore, ma limitato nella sua variazione.

In definitiva si nota ancora una volta la forte dipendenza dal numero dei cavalli nell'elaborazione della soluzione: al crescere del loro numero diventano particolarmente importanti il numero degli archi rispetto al numero dei nodi (Dijkstra e BFS), mentre la scelta di un algoritmo lineare nel numero dei nodi, cioè delle possibili posizioni, porta a risultati soddisfacenti, limitando il costo computazionale.

24 In questa sezione si riporta solo a titolo esemplificativo e di correttezza anche il tempo computazionale dell'algoritmo “Montagna” malgrado l'impossibilità di mostrare una sua correttezza a livello sperimentale se non stravolgendone i principi guida.