Robot Planning
Esplorazione di ambienti sconosciuti
Implementazione in C++

Alessandro Santuari

4 dicembre 2001

Un robot deve costruire una mappa completa di un ambiente sconosciuto, visto come un grafo non orientato e connesso. Per fare ciò esso deve esplorare ogni nodo ed ogni arco del grafo utilizzando il minor numero di mosse possibili. La penalità di un algoritmo di esplorazione eseguito sul grafo G consiste nel numero di mosse in eccesso (considerate nel caso pessimo) al *lower bound* $|E(G)|$ che devono essere effettuate per esplorare l'intero grafo.

Viene descritto e codificato un algoritmo d'esplorazione proposto da Panaite e Pelc la cui penalità è $O(|V(G)|)$ per ogni grafo.

Indice

1 Introduzione

Algoritmi per esplorare e mappare ambienti sconosciuti hanno varie applicazioni[1]: ottenere mappe di reti esistenti (es. reti informatiche, sistemi idraulici, caverne inesplorate) per le quali non ci sono mappe oppure quelle esistenti non sono più attendibili (es. dopo che alcuni collegamenti in una rete informatica sono caduti). Un altro tipo di applicazione è ottenere mappe di ambienti in continua evoluzione, come Internet o il World Wide Web. A causa della natura dinamica e distribuita di questi sistemi è spesso impossibile mantenerne una mappa completa e aggiornata. Comunque, se la rete non cambia troppo rapidamente, possiamo utilizzare un algoritmo di esplorazione efficiente per ottenere fotografie occasionali della rete.

Sono stati studiati due approcci principali al problema dell'esplorazione di ambienti sconosciuti. In uno di questi l'ambiente è visto come un particolare insieme di forme geometriche, cioè come un terreno con ostacoli poligonali [3] o rettangolari [4]. L'altro approccio consiste nel modellare l'ambiente come un grafo dove il robot può muoversi solo lungo gli archi.

Nel nostro caso, in particolare, l'ambiente verrà modellato come un grafo non orientato e connesso. Il robot non ha nessuna conoscenza dell'ambiente a priori. Viene lasciato e attivato su un nodo arbitrario del *grafo-ambiente*, e da questo inizierà l'esplorazione muovendosi lungo gli archi. Quando il robot percorre un arco, esplora entrambi i nodi che ne stanno alle estremità, e l'arco stesso. Durante l'esplorazione il robot prepara una mappa parziale che consiste in tutti i nodi e gli archi che ha già visitato. Esso etichetta i nodi esplorati e può riconoscerli facilmente quando vi ritorna. Il robot conosce quali siano gli archi inesplorati che partono dal nodo sul quale si trova, ma non può sapere dove portano.

Per lasciare un nodo può scegliere due modi:

- percorrere uno *specifico* arco esplorato;
- utilizzare un arco inesplorato *arbitrario* (scelto da un avversario).

Nella valutazione degli algoritmi consideriamo il caso pessimo, cioè quando le mosse lungo archi inesplorati siano le più "sfortunate" possibili. È immediato verificare che l'algoritmo di esplorazione necessita di almeno $\Omega(|E(G)|)$ mosse per mappare l'intero grafo. Questo *lower bound* può essere raggiunto per grafi euleriani da algoritmi *off-line* che conoscano a priori il grafo. Comunque, nel nostro caso un algoritmo d'esplorazione dovrà compiere più mosse, sia a causa della topologia del grafo (potrebbe non essere euleriano), sia a causa della carenza di informazioni che potrebbero indicare un metodo efficiente per l'esplorazione.

[1] Questi casi d'applicazione sono riportati in [2].

2 Scopo

Scopo di questo progetto è presentare alcuni algoritmi di esplorazione, considerando come parametro di complessità il numero di mosse che il robot deve eseguire prima di aver esplorato l'intero grafo anziché il tempo di elaborazione.

Una caratteristica del robot è quella di muoversi molto lentamente rispetto alla sua velocità di elaborazione, per questo motivo è importante cercare un algoritmo che permetta l'esplorazione nel minor numero di mosse possibili, anche a scapito della complessità computazionale.

In particolare vedremo come alcuni algoritmi immediati come la visita *DFS* ed il *Greedy* raggiungono penalità lineari nel numero degli archi, mentre dimostreremo che l'algoritmo proposto da *Panaite e Pelc* [1] è lineare nel numero dei nodi.

3 Terminologia

Quando parliamo di grafo, intendiamo sempre un grafo non orientato e connesso. Un algoritmo di esplorazione in esecuzione consiste in una serie di *mosse lungo archi*. Gli archi già percorsi vengono chiamati ESPLORATI mentre gli altri sono detti LIBERI. Un nodo è SATURO se tutti gli archi che partono da quel nodo sono esplorati, altrimenti è detto LIBERO. Il robot è bloccato sul nodo v se v è saturo. Una *mossa penalità* è una qualsiasi mossa effettuata lungo un arco già esplorato. La penalità di un algoritmo A che esegue su un grafo G, indicato con $\mathcal{P}_A(G)$, è data dal numero totale di *mosse penalità* considerato nel caso pessimo.

Le penalità prodotte da un algoritmo A sono lineari nel numero di nodi di un grafo se esiste una costante c tale che per ogni grafo G, $\mathcal{P}_A(G) \leq c \cdot |V(G)|$.

4 Algoritmi d'esplorazione

In questo documento vengono confrontati ed analizzati tre diversi algoritmi d'esplorazione.

In realtà essi sono abbastanza simili e utilizzano la stessa strategia di base: continuare a percorrere archi inesplorati finché ne esistono di disponibili e, quando il robot rimane bloccato, utilizzare una determinata strategia per tornare indietro su un nodo *libero* (cioè dove sono ancora presenti degli archi inesplorati).

Ci interesserà capire l'ordine di grandezza delle penalità di questi algoritmi. Sia $G = (V, E)$ un grafo non orientato e connesso, sia $n = |V(G)|$ e $m = |E(G)|$. Vedremo che l'algoritmo proposto da Panaite e Pelc commette penalità $O(n)$ mentre altri algoritmi hanno penalità $O(m)$. Notare che m può essere $O(n^2)$ nel caso di grafi molto densi.

4.1 Depth First Search (DFS)

Un semplice metodo per esplorare un grafo ci viene dato dalla strategia *depth-first search*. L'algoritmo d'esplorazione è definito nel seguente modo: finché il nodo corrente è libero, il robot percorre un arco incidente inesplorato. Quando rimane bloccato su un nodo, il robot si sposta verso il nodo libero visitato più *recentemente*, utilizzando un cammino minimo nel rispetto della parte già esplorata del grafo.

La ragione che causa l'inefficienza dell'algoritmo DFS è, in un certo senso, opposta a quella della strategia GREEDY.

In particolare, l'algoritmo DFS non utilizza informazioni sulla parte del grafo già esplorata, esso guida il robot guardando ciecamente al proprio passato e scegliendo sempre il nodo libero visitato più recentemente, anche quando si trova molto lontano dal nodo corrente.

Per dimostrare che le penalità della DFS non sono lineari nel numero di nodi del grafo, Panaite e Pelc hanno proposto un'altra classe di grafi per i quali l'algoritmo DFS non funziona bene.

Uno di questi grafi è rappresentato in Figura 1. In questo caso supponiamo che il robot parta dal nodo a_0 e che l'avversario conduca il robot attraverso questo cammino: $[a_0, b_0, a_1, b_1, \ldots, a_5, c_4, a_4, c_3, a_3, \ldots, a_0, d_0, a_1, d_1, \ldots a_5, e_4, a_4, e_3, a_3, \ldots, a_0]$

Notare che questo cammino esplora ogni arco una e una sola volta tranne gli archi $[b_0, b_5]$, $[d_0, d_5]$, $[c_4, c_5]$ e $[e_4, e_5]$.

Ora il robot è bloccato su a_0. Gli unici nodi liberi sono $\{b_5, c_5, d_5, e_5\}$ che sono stati visitati esattamente in quest'ordine. L'algoritmo DFS esplorerà i nodi liberi nell'ordine inverso (cioè partendo dal più recente) attraversando il grafo quattro volte!

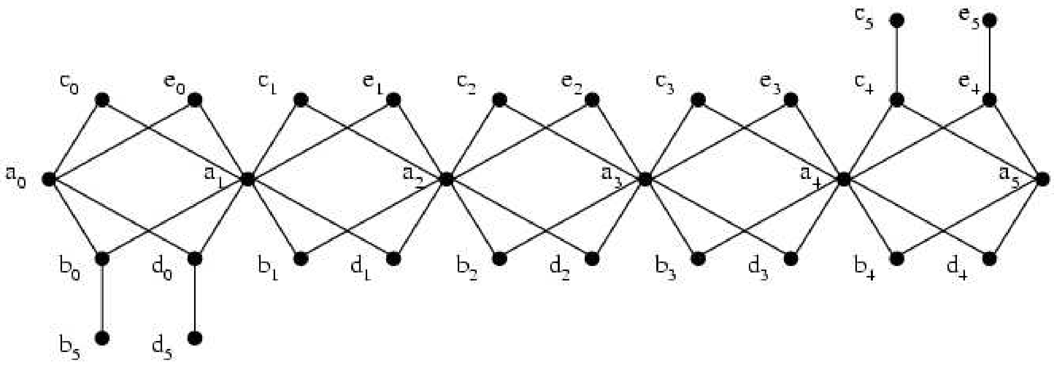


Figura 1: Esempio di grafo "cattivo" per la DFS

4.2 Greedy

L'algoritmo di esplorazione GREEDY è definito in questo modo: finché il nodo corrente è *libero*, il robot percorre un arco inesplorato che parte dal nodo. Quando rimane bloccato, il robot si muove verso il più *vicino* nodo libero, utilizzando un cammino minimo in accordo alla parte già esplorata del grafo.

La penalità dell'algoritmo GREEDY non è lineare nel numero di nodi del grafo. Per convicerci di ciò costruiremo dei grafi per i quali l'algoritmo GREEDY, sotto la guida di un avversario particolarmente cattivo, viene "tratto in inganno" e commette parecchie penalità.

Il modo per costringere il GREEDY a compiere tante mosse-penalità è il seguente: durante l'esplorazione, l'avversario crea dei nodi *liberi* che sono sufficientemente vicini al robot per "attrarlo" lontano da parti ancora inesplorate del grafo. Per poter tornare indietro a queste zone inesplorate, il robot dovrà compiere molte mosse-penalità.

Questi grafi speciali contengono particolari strutture chiamate *magneti* il cui scopo è proprio quello di trascinare lontano il robot da zone inesplorate.

Consideriamo un numero intero $g \geq 1$. Un *g-magnete* è un qualunque grafo isomorfo a $M_g = (V, E)$ con $V = \{0, \ldots, 4g\}$ e $E = \{[0,1], [1,2], \ldots, [4g-1, 0], [2g, 4g]\}$. Praticamente, un *g-magnete* è un ciclo di lunghezza $4g$ con collegato un nodo extra di grado 1. Definiamo inoltre:

- Il nodo con grado 3 è il *centro* del magnete.
- Il nodo con grado 1 è la *punta* del magnete.
- Il nodo alla distanza $2g$ dal centro è la *radice* del magnete.

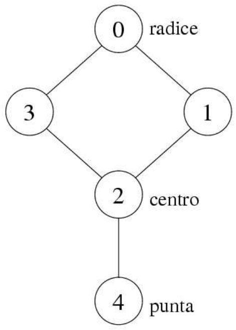


Figura 2: 1-magnete

Un magnete si collega al resto del grafo tramite il nodo *radice*. A seconda degli archi esplorati possiamo dire che un magnete è:

- *attivo* se il centro è stato visitato, ma la punta no.
- *morto* se la punta è stata esplorata.
- *addormentato* altrimenti.

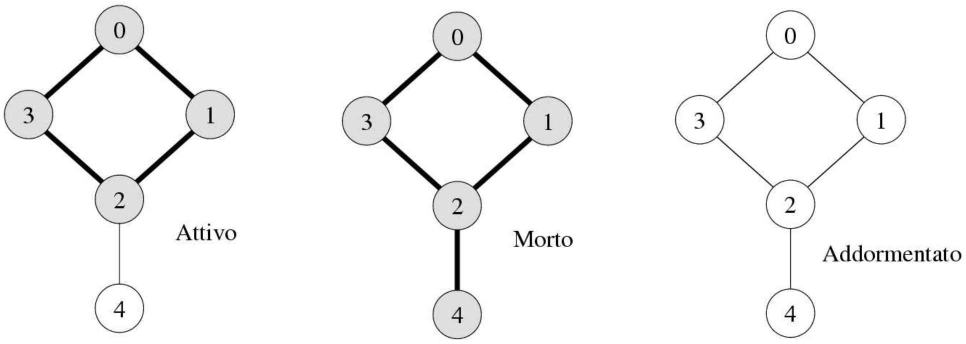


Figura 3: Stati di un magnete

È dimostrato che è possibile creare una classe di grafi contenenti strutture come i magneti, in grado di mettere in difficoltà l'algoritmo Greedy. L'esecuzione su questi grafi comporta sempre penalità molto alte, nell'ordine di $|E(G)|$. La dimostrazione completa si trova in [1].

In Figura 4 abbiamo un esempio di questi grafi: supponiamo di partire con il robot nel nodo b, inizialmente tutti i magneti sono *addormentati*. A questo punto il Greedy si muoverà sempre lungo archi inesplorati, finché possibile. L'avversario guida il robot lungo gli archi disegnati in nero arrivando infine nel nodo a. Notare che grazie alle mosse effettuate dall'avversario, tutti i magneti sono *attivi*.

Ora il robot si muoverà nell'unico nodo inesplorato adiacente ad a e qui si blocca. Notare che fino a questo punto il robot non ha commesso alcuna penalità. A questo punto però la "trappola" è completa!

Il robot che segue l'algoritmo Greedy si muoverà di nuovo verso b, uccidendo tutti i 0-magneti lungo la strada (questi movimenti comportano molte penalità). Una volta in b ucciderà anche l'1-magnete radicato in b. Ma attenzione! Il robot è stato attratto dai magneti molto lontano dall'unico nodo da esplorare, la punta dell'1-magnete radicato in a. Ora dovrà compiere molte penalità per tornare indietro!

Panaite e Pelc hanno proposto un metodo ricorsivo per creare grafi di questo tipo, sempre più complessi e sempre più "pericolosi" per il Greedy.

Un altro esempio di grafo è riportato in Figura 5. Il procedimento è lo stesso ma qui il robot compie ancora più penalità. Infatti dopo il viaggio condotto dall'avversario, il robot si muove da a verso b uccidendo tutti gli 0-magneti, poi torna ad a uccidendo stavolta tutti gli 1-magneti sulla via. Qui uccide il 2-magnete radicato in a. Notare che il robot dovrà ancora spostarsi in b per uccidere l'ultimo magnete rimasto!

Ora che abbiamo visto questi esempi possiamo riflettere sul motivo che causa l'inefficienza del Greedy: il robot si muove sempre cercando il più vicino nodo libero, senza preoccuparsi se questi spostamenti causeranno molte penalità in futuro.

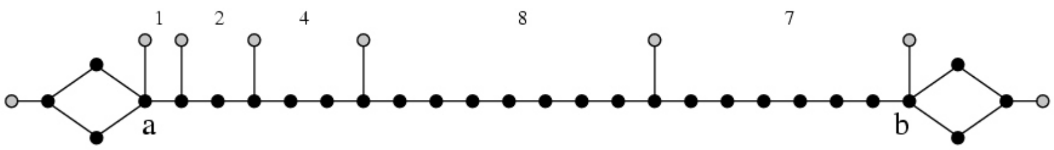


Figura 4: Esempio n.1 di grafi con magneti

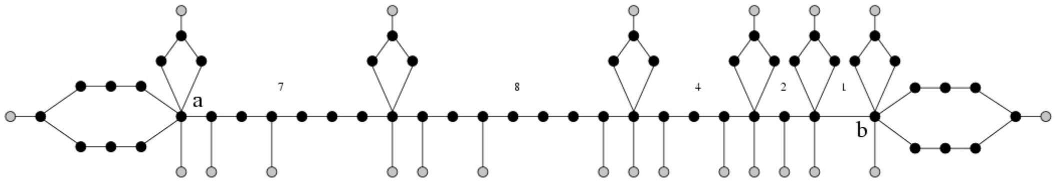


Figura 5: Esempio n.2 di grafi con magneti

4.3 L'algoritmo di Panaite e Pelc

Questo algoritmo non commette mai più di $3|V(G)|$ mosse-penalità per ogni grafo connesso G. L'idea principale è quella di commettere la maggior parte delle penalità su un albero di copertura costruito dinamicamente. Questo algoritmo si basa su due funzioni:

4.3.1 Esplora

Questa funzione controlla i movimenti del robot sull'albero di copertura a partire dal nodo r. Al termine di questa funzione l'intero grafo è esplorato e il robot si trova sul nodo di partenza.

In questa funzione si usano due array: $parent : V(G) \rightarrow V(G) \cup \{null, unassigned\}$ e $color : V(G) \rightarrow \{white, blue, red\}$.

Per ogni $v \in V(G) \backslash \{r\}$, $parent[v]$ rappresenta il padre di v nell'albero di copertura T di G che ESPLORA(r) costruisce nella sua esecuzione.

ESPLORA(r)

$parent[r] := null$;
$parent[u] := unassigned \quad \forall u \in V(G) \backslash \{r\}$;
$color[u] := white \quad \forall u \in V(G)$;
$v := r$;
while $v \neq null$ **do**
    SATURA(v);
    if v ha un vicino u tale che $parent[u] = unassigned$
    then
        $parent[u] := v$;

$v := u;$

else

$v := parent[v];$

MuoviSu(v);

La procedura Satura(v) è molto importante e sarà descritta in seguito. Il suo scopo è quello di far viaggiare il robot partendo ed infine tornando al nodo v, inoltre v diventa saturo.

Per definizione della Satura, ogni volta che si giunge alla condizione **if**, il robot conosce tutti i vicini di v. Successivamente l'algoritmo guida il robot sull'albero di copertura e alla fine tutti i nodi saranno visitati e saturati, quindi l'intero grafo sarà esplorato. Quando l'albero di copertura risulta completo, il robot lo risale tornando al nodo r.

Durante l'esecuzione di Esplora, possiamo dividere le *mosse-penalità* in due tipi:

- **Interne**: quelle richieste dalla procedura Satura;
- **Esterne**: quelle commesse dalla procedura Esplora. Notare che avendo prima saturato il nodo v, ogni mossa di Esplora comporta una penalità. Possono essere ulteriormente suddivise in penalità di *crescita arborescenza*, commesse nel ramo **then**, e di *backtrack* generate dal ramo **else**.

Possiamo notare come le *mosse-penalità* esterne sono lungo gli archi dell'albero di copertura generato dinamicamente. Visto che l'algoritmo muove il robot nell'albero di copertura in avanti secondo l'ordine DFS e poi torna indietro, possiamo rilevare il seguente risultato parziale sulla complessità:

Lemma 1 *Per ogni grafo connesso G, $\mathcal{P}_{ESPLORA}(G) \leq 2(|V(G)| - 1) + p$, dove p è il numero di* mosse-penalità *commesse complessivamente dalla procedura* Satura.

Supponiamo inizialmente che il grafo G sia euleriano. In questo caso la procedura Satura(v) può semplicemente guidare il robot lungo archi inesplorati finché il nodo corrente e' *libero*. Visto che il grafo è euleriano, ogni nodo ha grado *pari* e il viaggio iniziato da v termina anche in v. Quindi non viene commessa alcuna *penalità interna* dalla procedura Satura. Chiamiamo ora Esplora' l'algoritmo di Esplora funzionante solo per grafi euleriani e che utilizza la versione della Satura vista sopra. Per ogni grafo euleriano G abbiamo che $\mathcal{P}_{ESPLORA'}(G) \leq 2|V(G)|$.

Consideriamo ora il caso generale. Il cammino ottenuto percorrendo sempre archi inesplorati non termina necessariamente al nodo di partenza. Per poter saturare v e poi riportare il robot sul nodo di partenza, la Satura ha bisogno di eseguire una sequenza di *cammini-liberi* e mantenere un *ponte* tra il nodo corrente e il nodo di partenza v.

Un *cammino-libero* è la massima sequenza di mosse consecutive lungo archi inesplorati. Durante l'esecuzione della Satura, alcuni nodi *bianchi* divengono *blu*. Un *ponte* è un qualsiasi cammino nella parte esplorata di G che utilizza solo nodi blu.

Vengono commesse mosse penalità solo quando il robot deve spostarsi tra due cammini-liberi.

4.3.2 Satura

SATURA(v)

Lo scopo di questa funzione è quello di saturare il nodo v, cioè esplorare tutti gli archi che partono da v. Al termine di questa funzione il robot si trova ancora in v e v è saturo.

SATURA(v)

if v è saturo **then** return;
color[v] := *blue*;
$u := v$;
while NOT ($u = v$ AND v è saturo) **do**
  if $\exists$ un arco inesplorato e **then**
    u :=l'altro capo di e;
    color[u] := *blue*;
  else
    Sia $[u = u_0, u_1, \ldots, u_k = v]$ un cammino da u a v dove
      tutti i nodi sono *blu* e tutti gli archi esplorati;
    color[u] := *red*;
    $u := u_1$;
  MUOVISU(u);
Tutti i nodi *blu* tornano *bianchi*;

Dalla definizione della procedura SATURA possiamo enunciare i seguenti lemmi:

Lemma 2 *La procedura* SATURA*(v) è ben definita.*

Lemma 3 *La procedura* SATURA*(v) termina quando il nodo v è saturo e il robot è in v.*

Lemma 4 *Per ogni esecuzione della* SATURA*, il numero di mosse-penalità è uguale al numero di nodi* blu *cambiati in* rosso*.*

Lemma 5 *Un nodo* rosso *non cambierà il suo colore.*

Dimostrazione:
(2). È sufficiente dimostrare la seguente proprietà $\forall i \geq 1$

P(i): se SATURA(v) esegue l'*i-esima* iterazione del ciclo `while` allora, prima di questa iterazione, esiste un *ponte* che collega u a v.

$P(1)$ è vera. Supponiamo che $P(i)$ sia vera per qualche $i \geq 1$ e che SATURA(v) stia eseguendo l'*(i+1)-esima* iterazione del ciclo while.

Sia u il nodo dove il robot si trova prima della *i-esima* iterazione. Per ipotesi di induzione, esiste un cammino B nella parte esplorata di G formato solamente da nodi *blu* che collega u a v.

- Se u non è saturo, sia u' il nodo scelto dalla procedura SATURA. Questo nodo sarà colorato di *blu* e prima dell' *(i+1)-esima* iterazione, l'arco $[u', u]$ seguito dal cammino B è un ponte che collega u' a v

- Se u è saturo, sia $[u_0, u_1, \ldots, u_k]$ il cammino definito dalla SATURA. Questo cammino è esattamente il ponte B. Quindi, prima dell' *(i+1)-esima* iterazione il cammino $[u_1, \ldots, u_k]$ è un ponte che connette u_1 a v.

Perciò, in entrambi i casi, $P(i+1)$ è vera.

(3). Dopo ogni iterazione nel ciclo while, o viene esplorato un nuovo arco, o un nuovo nodo diviene *rosso*. SATURA colora un nodo di *rosso* una volta sola. Notare che al di fuori della procedura SATURA i nodi non cambiano colore. Visto che il numero di archi da esplorare e il numero di nodi da colorare in *rosso* sono finiti, ci sarà un'iterazione dove si torna sul nodo v ed esso sarà saturo.

(4). Una massima sequenza di iterazioni del ciclo while, con u non saturo, definisce un cammino-libero per il robot. L'unica mossa penalità è commessa nella SATURA quando il robot si muove da u a u_1. Questa condizione si verifica nel ramo **else**, dove si colora il nodo di *rosso*.

(5). Un nodo viene colorato di *rosso* solo quando è saturo. Infatti se non lo fosse, esisterebbe almeno un arco inesplorato, e non sarebbe eseguito il ramo **else**. Una volta che un nodo è diventato *rosso*, il suo colore non può essere cambiato in quanto la SATURA non muoverà mai più il robot su questo nodo. Notare inoltre che all'esterno della procedura SATURA i nodi non cambiano colore.

Teorema 1 *La penalità dell'algoritmo* ESPLORA *è lineare nel numero di nodi del grafo.*

Dimostrazione:
Proveremo che, per ogni grafo connesso G, $\mathcal{P}_{ESPLORA}(G) \leq 3|V(G)|$. Per il Lemma 1, è sufficiente dimostrare che il numero di mosse-penalità commesse da *tutte* le esecuzioni di SATURA, non supera $|V(G)|$.

Questo segue direttamente dai Lemmi 4 e 5.

4.3.3 Possibili miglioramenti

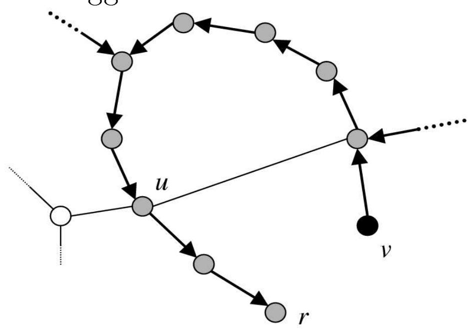
Le penalità di questo algoritmo sono necessariamente comprese tra $2|V|$ e $3|V|$ in quanto si deve almeno generare l'albero di copertura e visitarlo. Infatti avremo sempre $2(|V|-1)$ penalità esterne: precisamente $|V|-1$ penalità per creare l'albero (*prima* si satura il nodo e poi si prosegue) e altre $|V|-1$ penalità risalendo l'albero di copertura. Possiamo identificare due punti di debolezza dell'algoritmo proposto da Panaite e Pelc:

- Viene *sempre* commessa una penalità esterna al termine della SATURA. Infatti, quando v è saturo, il robot deve necessariamente eseguire una penalità se vuole allontanarsi da v!
- Quando il robot rimane bloccato su v e deve tornare indietro, esso risale "ciecamente" l'albero di copertura passo passo anche se si potrebbero utilizzare cammini più brevi.

Per aggirare il primo problema possiamo modificare la funzione SATURA in modo che termini quando il nodo è saturo *oppure* quando esiste un solo arco inesplorato incidente. Chiameremo questa nuova funzione QUASISATURA.

Invece, un metodo per affrontare il secondo problema consiste nel modificare la funzione ESPLORA in questo modo: quando il robot è bloccato sul nodo v, *prima* di muoversi analizza la parte dell'albero di copertura già creata per trovare il primo nodo u (lungo il cammino $v-r$) per il quale è possibile continuare la costruzione dello spanning tree. Dopodiché il robot si sposterà verso u seguendo un *cammino minimo* da v a u che, in generale, comporta meno penalità di risalire l'albero fino al nodo u. Si veda Figura 6.

Figura 6: Vantaggio nel ritorno con cammino minimo.



Per comodità definiamo tre nuove funzioni:

- NODOCOMPLETO(V) che restituisce *vero* se tutti i nodi adiacenti a v fanno già parte dell'albero di copertura.
- POSIZIONE() che restituisce il nodo sul quale si trova il robot.

- MuoviCamminoMinimo(v) che sposta il robot dalla posizione corrente al nodo v utilizzando un cammino minimo.

```
bool NodoCompleto(v)
for each u ∈ Adiacenza(v) do
    if parent[u] = unassigned;
    then return false;
return true;
```

```
Esplora2(r)
parent[r] := null;
parent[u] := unassigned   ∀u ∈ V(G)\{r};
color[u] := white   ∀u ∈ V(G);
v := r;
while v ≠ null do
    QuasiSatura(v);
    if v non è saturo
    then // soluzione al problema 1
        MuoviSuInesplorato();
        u := Posizione();
        parent[u] := v;
        v := u;
    else
        if v ha un vicino u tale che parent[u] = unassigned
        then
            parent[u] := v;
            v := u;
        else // soluzione al problema 2
            n := v;
            while parent[n] ≠ null AND NodoCompleto(n) = false do
                n := parent[n];
            MuoviCamminoMinimo(n);
            v := n;
        MuoviSu(v);
```

In questo modo, quando possibile, per risolvere il primo problema si sceglie direttamente l'arco da inserire nell'albero di coperura senza commettere penalità. Se u è il nodo collegato all'unico arco inesplorato, *parent*[u] sarà certamente *unassigned*, in quanto non è stata ancora eseguita QuasiSatura(u).

5 Implementazione

Il programma è stato sviluppato in C++ ed è in grado di simulare il comportamento dei tre algoritmi sopra esposti. L'utente può caricare un grafo da file oppure utilizzare il generatore casuale di grafi presente nel programma. Inoltre può visualizzare il grafo e preparare un immagine per ogni passo dell'algoritmo.
Dopo aver eseguito `make` lanciare l'eseguibile `Robot`.

5.1 Caricare un grafo

Per caricare un nuovo grafo, utilizzare l'opzione `Cambia grafo`. Ora potrete scegliere se caricare un grafo da file o generarlo casualmente.

- **Carica da file:** Dopo aver selezionato `Carica da file` digitare il nome del file. Il file deve rispettare questo formato:
  ```
  NumNodi
  N1 N2
  N3 N4
  .  .
  ```

- **Genera casualmente:** Selezionare l'opzione `Genera casualmente`. A questo punto inserire il numero di nodi ed il numero di archi desiderati. Notare che se $n = |V|$ e $m = |E|$, deve essere che: $n - 1 \leq m \leq n(n-1)/2$
  Infatti il grafo deve essere connesso e il numero di archi non può superare quello del grafo completo.

  Nota: Come vengono generati i grafi casuali.

  Supponiamo di dover generare un grafo connesso G con $|V| = n$ e $|E| = m$. Il programma segue questi passi:

  1. Crea n nodi.
  2. Scegli due nodi a caso e collegali con un arco. Inserisci questi nodi nella parte connessa.
  3. Collega un nodo appartenente alla componente connessa e uno non appartenente. Inserisci il secondo nodo nella parte connessa di G.
  4. Ripeti il punto 3 fino a raggiungere un'unica componente connessa (abbiamo generato un albero random).
  5. Continua ad aggiungere archi fra nodi scelti casualmente fino a raggiungere m desiderato.

Se il grafo è stato caricato correttamente apparirà una linea indicante il numero di nodi e il numero degli archi.

5.2 Scegliere l'algoritmo

All'avvio del programma è selezionato l'algoritmo proposto da Panaite e Pelc. Per cambiare l'algoritmo utilizzato, scegliere l'opzione `Cambia algoritmo`. Gli algoritmi disponibili sono *Greedy*, *DFS*, *Panaite&Pelc* e *Panaite&Pelc (migliorato)*.

5.3 Modalità avversario

Abbiamo supposto in precedenza che le mosse del robot lungo gli archi inesplorati venivano decise da un "avversario" che cercava di far compiere più mosse penalità al robot.
Normalmente l'utente dovrebbe impersonare l'avversario e scegliere l'arco inesplorato da percorrere. Questo potrebbe diventare noioso per grafi abbastanza grandi, perciò come impostazione di default la scelta dell'arco viene effettuata in maniera casuale dal programma.

Attenzione! Randomizzando la scelta dell'arco non potremo valutare gli algoritmi nel caso pessimo, infatti le scelte casuali sono ben diverse dalle scelte di un avversario malizioso.

5.4 Visualizza grafo

Viene utilizzato il pacchetto gratuito `Graphviz` [2] della AT&T per visualizzare il grafo caricato in memoria! Verranno presentate due immagini realizzate da due programmi differenti. Le immagini sono in formato PS e possono essere stampate facilmente per avere un riferimento del grafo in questione.

5.5 Modalità scatta-fotografie

Durante l'esecuzione dell'algoritmo è possibile tenere traccia dei passi del robot salvando numerose immagini. Purtroppo creare un fotografia della situazione è un operazione dispendiosa, per questo la modalità `scatta-foto` è disabilitata di default.

5.6 Esegui

Una volta selezionato il grafo e l'algoritmo da utilizzarsi, e dopo aver specificato le altre opzioni siamo pronti all'esecuzione vera e propria dell'algoritmo. Se è stata attivata l'opzione `scatta-fotografie` questa fase sarà molto lenta. Inoltre se è stata attivata l'opzione `avversario manuale`, ogni volta che il robot decide di muoversi lungo archi inesplorati verrà stampata una lista di destinazioni possibili. L'utente che svolge il ruolo di avversario deve indicare il nodo scelto.

[2] http://www.research.att.com/sw/tools/graphviz

Terminata l'esecuzione, verrà stampato il numero di mosse totali e il numero di mosse penalità effettuate.

6 Prove sperimentali

6.1 Esempi di esecuzione

Per chiarire meglio il funzionamento degli algoritmi è utile ricorrere ad esempi. Di seguito abbiamo delle possibili esecuzioni dei tre algoritmi sullo stesso grafo. Supponiamo che il robot parta sempre dal nodo **1** e che l'avversario compia scelte casuali.

6.1.1 Esempio DFS

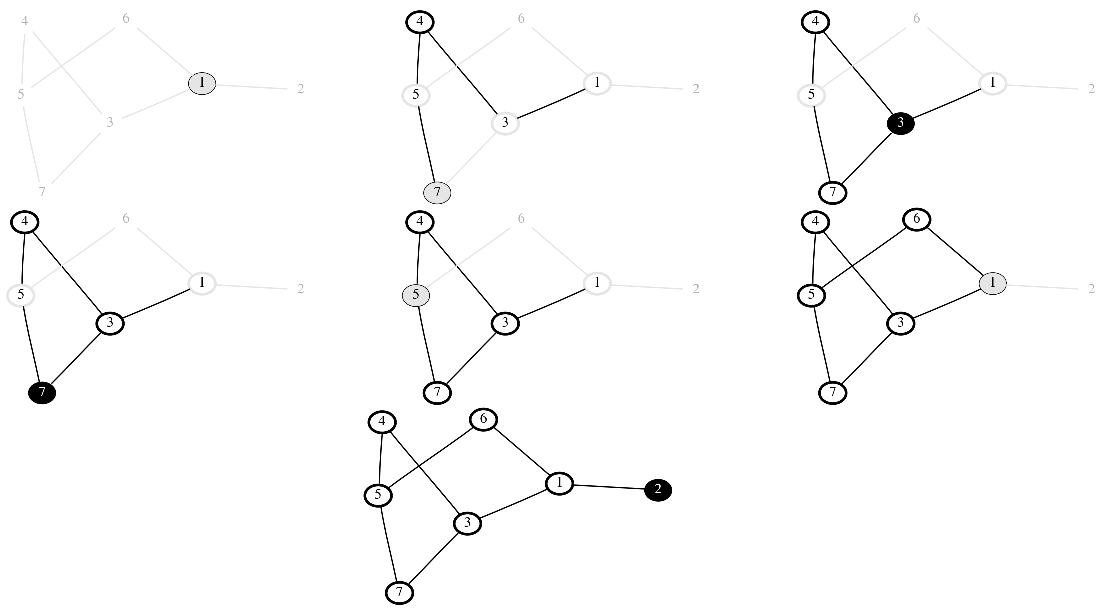


Figura 7: Esempio esecuzione DFS

La simulazione si trova in Figura 7. Inizialmente il robot segue le mosse dell'avversario fino a quando si blocca sul nodo **3**. Ora l'algoritmo DFS esegue un *backtrack* fino a tornare al nodo libero visitato più recentemente, nel nostro caso il nodo **5**. Da qui segue l'unico percorso libero per completare l'esplorazione del grafo.
Mosse: 10 Penalità: 2

6.1.2 Esempio Greedy

L'esempio dell'algoritmo greedy si trova in Figura 8. Inizialmente le mosse non differiscono dalla DFS (supponiamo che l'avversario abbia generato la stessa sequenza

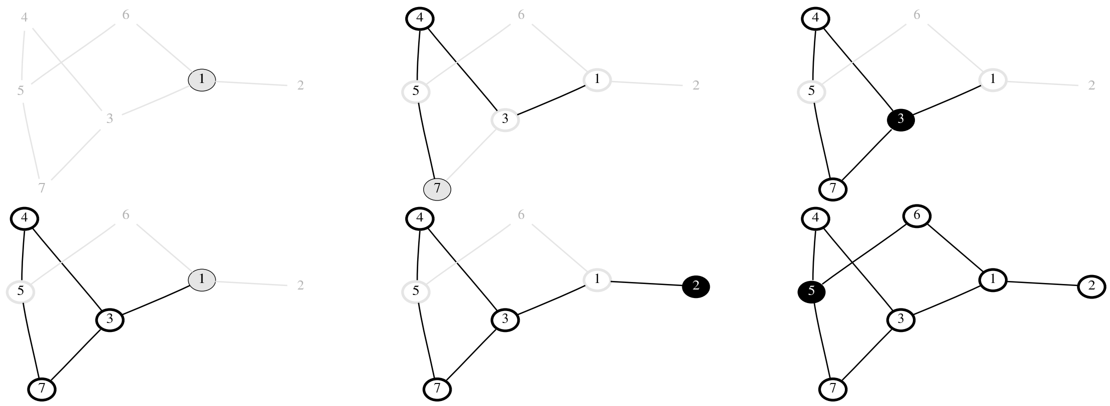


Figura 8: Esempio esecuzione Greedy

di mosse) e il robot prosegue fino a che non si blocca su **3**. Ora l'algoritmo Greedy cercherà di muovere il robot sul più vicino nodo libero, cioè il nodo **1**. L'esplorazione prosegue scegliendo archi inesplorati. Nel nostro caso il robot si blocca ancora su **2** prima di terminare l'esplorazione.
Mosse: 10 Penalità: 2

6.1.3 Esempio Panaite&Pelc

L'algoritmo di Panaite e Pelc agisce in maniera più complicata. Anche se sappiamo che questo algoritmo non commetterà mai più di $3|V| = 3 \cdot 7 = 21$ penalità ci accorgiamo di quanto questa limitazione sia alta, anche vedendo le figure necessarie per mostrare l'esecuzione.

Il nodo **1** diventa la radice dell'albero di copertura creato dinamicamente dall'algoritmo di esplorazione. Inizialmente si esegue la Satura del nodo **1**: il robot percorre archi inesplorati finché non ha saturato **1** o non si blocca. Dopo alcune mosse infatti il robot si blocca in **2**. Da qui deve tornare al nodo **1** utilizzando un ponte, cioè un cammino con solo nodi blu. Una volta in **1** la procedura Satura termina e il controllo torna alla procedura Esplora.

Ora viene scelto un vicino arbitrario di **1**, nell'esempio il nodo **6**. L'arco $(1, 6)$ ora fa parte dell'albero di copertura. Il nodo **6** è già saturo, perciò il robot continua verso **5** inserendo $(5, 6)$ nell'albero di copertura. Ora si esegue Satura(5). Al termine di essa il robot si trova ancora in **5** e questo nodo è saturo.

Il procedimento continua e nell'albero di copertura vengono inseriti gli archi $(5, 7)$, $(3, 7)$, $(3, 4)$. Ora il robot si trova in **4** ma la procedura Esplora non può continuare l'albero in quanto il nodo **5** è già stato utilizzato. Deve tornare indietro lungo l'albero che l'algoritmo ha costruito fino a questo momento fino a trovare un nodo per il quale è possibile continuare la costruzione dello spanning tree. Nel nostro caso il robot torna a **1** utilizzando il cammino $[4, 3, 7, 5, 6, 1]$ compiendo molte *penalità esterne*.

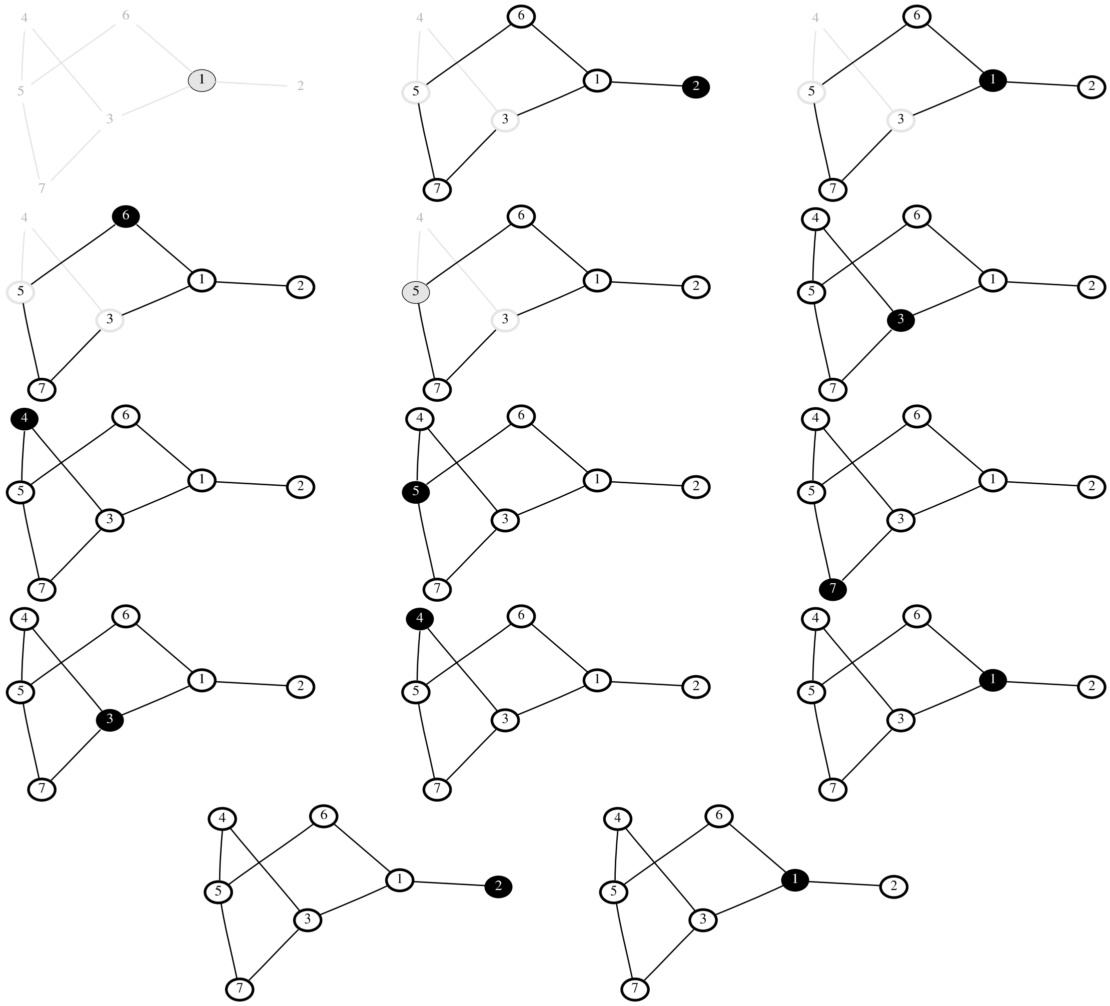


Figura 9: Esempio esecuzione Panaite&Pelc

Ora manca solo l'arco $(1, 2)$ per completare l'albero di copertura. Il robot si muove in **2**, esegue qui la SATURA che termina subito.

Ora la procedura ESPLORA termina, in quanto l'albero è completo e su ogni nodo è stata eseguita la SATURA.

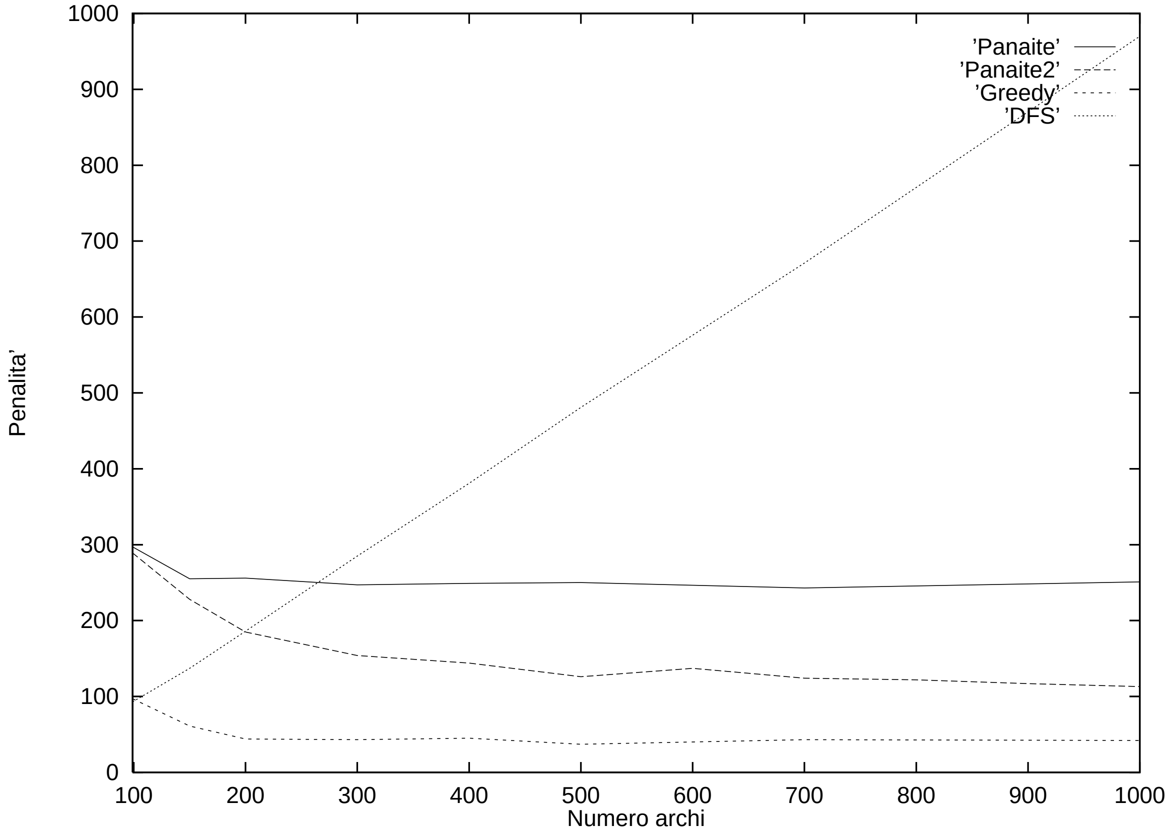
Mosse: 22 Penalità: 14 (Interne: 3, Esterne: 12)

6.2 Verifica

Utilizzando il programma `Robot` verifichiamo i risultati di questi tre algoritmi su grafi generati casualmente.

Per iniziare, poniamo $|V| = 100$ e facciamo variare $|E|$. La scelta dell'arco inesplorato da percorrere viene effettuata casualmente. All'aumentare del numero di archi le penalità generate dagli algoritmi *Depth-First Search* e *Greedy* dovrebbero aumentare mentre le penalità generate dall'algoritmo di *Panaite e Pelc* dovrebbero rimanere costanti. Il risultato dell'esperimento si trova in Figura 10, in ascisse abbiamo il numero di archi mentre in ordinata le penalità effettuate. L'algoritmo di Panaite e Pelc si comporta come avevamo previsto: all'aumentare di $|E|$ le penalità non aumentano. Possiamo verificare il teorema $\mathcal{P}_{ESPLORA}(G) \leq 3|V(G)|$, infatti le penalità non superano mai $3|V(G)| = 300$ anche se si avvicina molto a questo valore.

Figura 10: $|V| = 100$



La versione modificata di questo algoritmo è migliore, infatti non abbiamo più una limitazione inferiore alle penalità esterne[3]. In particolare si vede che all'aumentare della densità del grafo, le modifiche diventano sempre più rilevanti. È soprattutto la modifica del "ritorno" che trae vantaggio dalla densità del grafo. Infatti maggiori sono gli archi disponibili, più diventa facile trovare un cammino breve per risalire l'albero di copertura.

[3]Nell'algoritmo Panaite e Pelc sono *sempre* $2(|V| - 1)$.

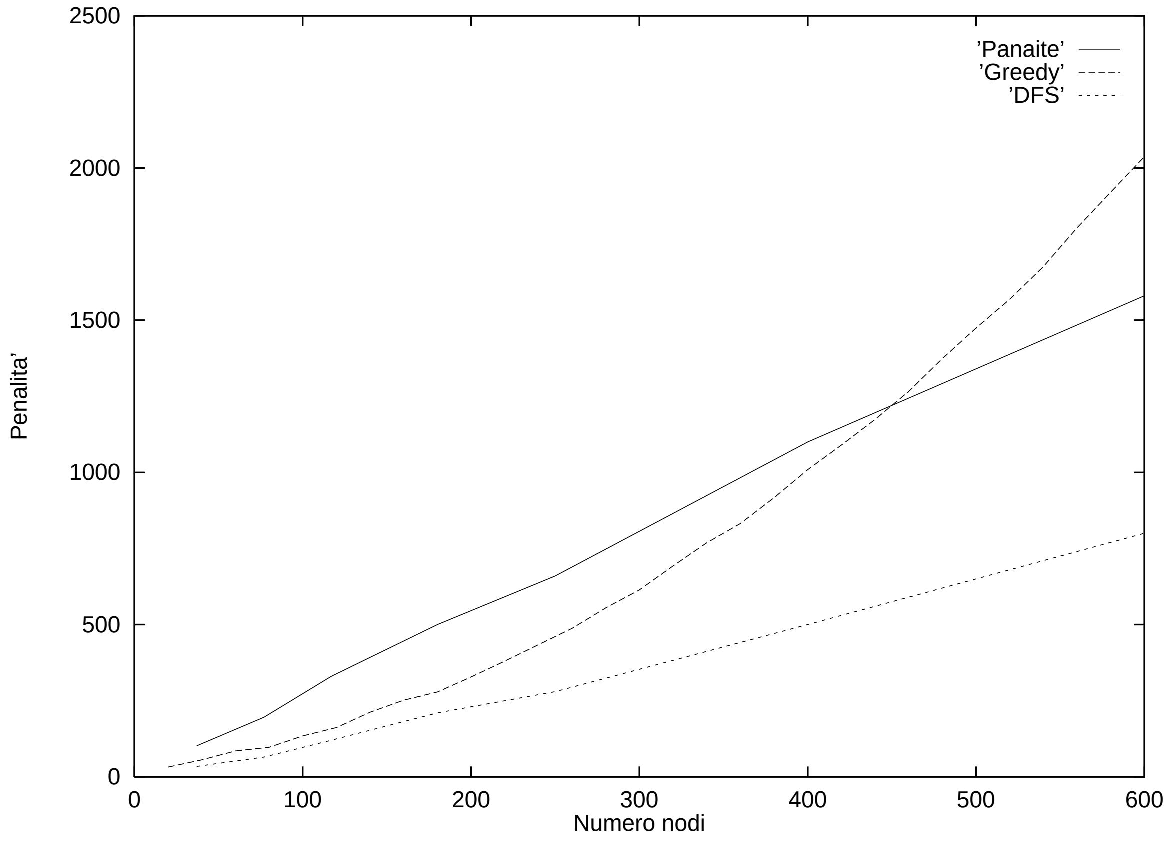
Dal grafico si vede bene come l'algoritmo DFS generi mosse penalità lineari in $|E|$, proprio come avevamo previsto.

Invece l'andamento dell'algoritmo Greedy non è conforme alle aspettative. Dal grafico sembra costante in $|E|$, ed addirittura migliore di Panaite e Pelc se consideriamo le costanti!

Questo perché utilizzando grafi ed avversari casuali la probabilità che l'algoritmo debba affrontare il caso pessimo è molto bassa. In pratica, abbiamo verificato sperimentalmente la correttezza della complessa costruzione proposta da Panaite e Pelc.

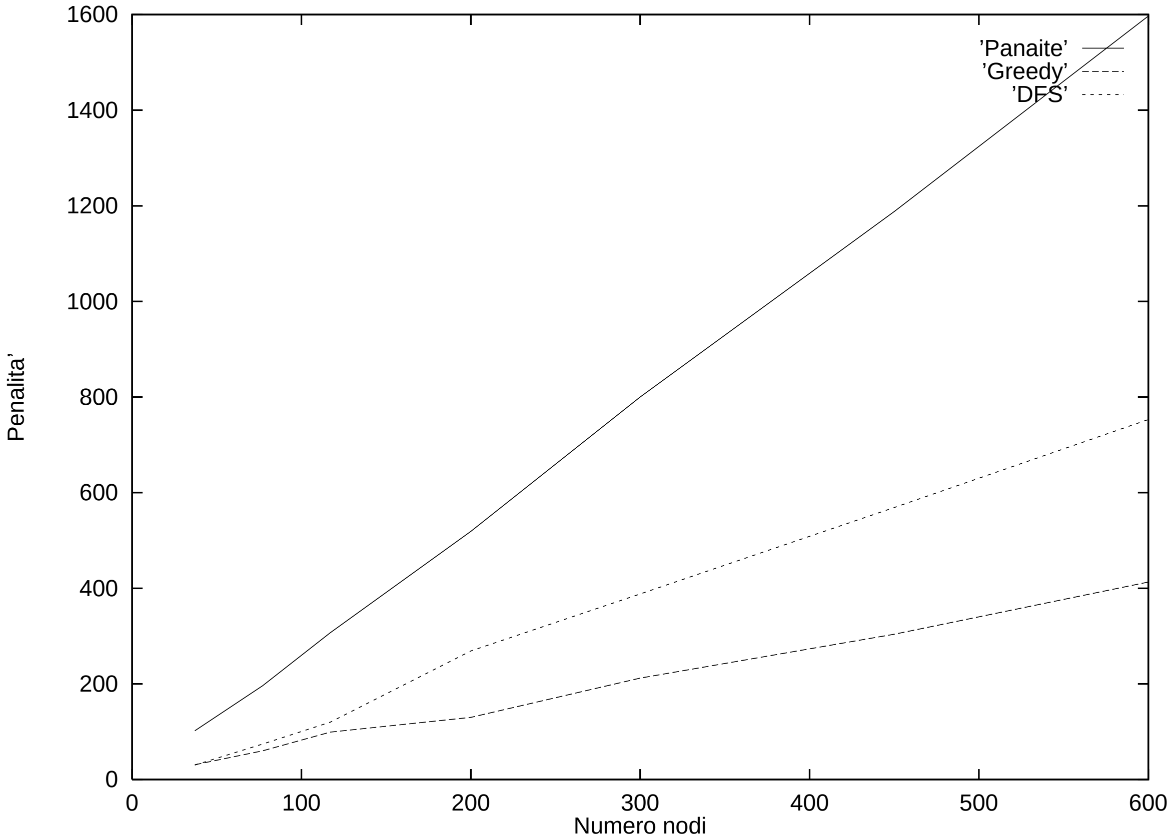
Proviamo a fare un'altra prova con grafi sfavorevoli al Greedy. Inoltre, per valutare il caso pessimo utilizziamo un avversario malizioso. In questo caso vediamo come l'algoritmo Greedy sia, nel caso pessimo, non lineare nel numero di nodi. L'ultimo

Figura 11: Grafi sfavorevoli per Greedy, avversario malizioso



esperimento consiste nell'esplorare questi stessi grafi utilizzando una scelta casuale dell'arco inesplorato. I risultati sono in Figura 12. In questo caso GREEDY produce il minor numero di penalità nonostante i grafi siano "sfavorevoli". In verità, se i magneti non vengono attivati correttamente, questi grafi non creano problemi all'algoritmo Greedy.

Figura 12: Grafi sfavorevoli per Greedy, avversario "casuale"



6.3 Conclusioni

- **DFS**. Abbiamo visto come l'esplorazione DFS non permette una buona efficienza in quanto le penalità crescono sempre come $|E|$. Dal punto di vista delle penalità generate, questo algoritmo è il peggiore dei tre algoritmi analizzati.

- **Panaite&Pelc**. L'efficienza dell'algoritmo proposto da Panaite e Pelc è certamente superiore se consideriamo il caso pessimo. Le sue penalità infatti non superano mai $3|V|$ indipendentemente da $|E|$. Nonostante ciò, il rapporto penalità/mosse effettuate diventa inaccettabile se i grafi sono piccoli o sparsi. Ad esempio, come vediamo da Figura 10, l'algoritmo Panaite eseguito su un grafo con $|V| = 100$ e $|E| = 99$ (cioè un albero) produce penalità pari a 300, mentre ad esempio il Greedy ne commette solo 100.

In definitiva è preferibile utilizzare questo algoritmo per grafi piuttosto densi e quando si vuole avere la certezza di non commettere più di $3|V|$ penalità. Purtroppo nell'esplorazione di grafi sconosciuti non possiamo dire se il grafo è sparso o denso, né tanto meno conoscere il numero di nodi del grafo.

- **Panaite&Pelc Migliorato**. Pur rimanendo lineare nel numero di nodi, quest'algoritmo riesce ad abbattere la limitazione inferiore delle penalità esterne portando di fatto la complessità da $\Theta(|V|)$ a $O(|V|)$. Questo ci consente un'esplorazione efficiente in pratica e anche se le penalità totali sono più alte di quelle dell'algoritmo Greedy abbiamo un importante vantaggio: la garanzia di non commetterne mai più di $3|V|$. Inoltre le prestazioni di questo algoritmo aumentano per grafi densi.

- **Greedy**. L'algoritmo Greedy si è rivelato essere sempre il migliore in tutte le prove sperimentali effettuate con grafi casuali e avversario automatico. Purtroppo però abbiamo dimostrato che Greedy non è lineare nel numero di nodi costruendo una famiglia di grafi di controesempio. I risultati di queste prove si trovano in Figura 11. Questi grafi (e particolari movimenti dettati dall'avversario) costituiscono il caso pessimo del greedy.

  In pratica però possiamo dire che l'algoritmo Greedy è il migliore algoritmo di esplorazione, in quanto la probabilità che si presenti un grafo sfavorevole e soprattutto che tutte le mosse effettuate dal robot coincidano con quelle del caso pessimo è praticamente zero. Basta ad esempio randomizzare la scelta dell'arco inesplorato da percorrere per ridurre drasticamente la probabilità che il robot percorra esattamente la strada pessima.

  Abbiamo infatti verificato sperimentalmente che i grafi sfavorevoli al Greedy proposti da Panaite e Pelc si comportano come un grafo normale se il robot non percorre il particolare cammino iniziale mostrato alla sezione 4.2 (si veda il grafico in Figura 12).

Dopo aver comparato questi tre algoritmi possiamo dire che il migliore algoritmo d'esplorazione di grafi sconosciuti non orientati è, in pratica, l'algoritmo Greedy.

Se però vogliamo avere una limitazione superiore alle penalità è preferibile utilizzare l'algoritmo proposto da Panaite e Pelc (possibilmente nella sua versione modificata), anche se produce in media maggiori penalità.

Riferimenti bibliografici

[1] P. Panaite e A. Pelc. Exploring Unknown Undirected Graphs. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1998.

[2] M. Bender, A. Fernández, D. Ron, A. Sahai e S. Vadhan. The power of a pebble: Exploring and mapping directed graphs. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 269-278, Maggio 1998.

[3] E. Bar-Eli, P. Berman, A. Fiat e R. Yan. On-line navigation in a room. *J. Algorithms 17*, pages 319-341, 1994.

[4] X. Deng, T. Kameda e C. H. Papadimitriou. How to learn an unknown enviroment, the rectilinear case. *J. ACM 45*, pages 215-245, 1998.