SAPIENZA
UNIVERSITÀ DI ROMA

Strumento per la verifica di proprietà su diagrammi Gantt e Pert attraverso l'analisi formale

tesina di

Metodi Formali nell'Ingegneria del Software

Prof. Toni Mancini

Realizzata da:

Mino Germano

Massimo Bonotti

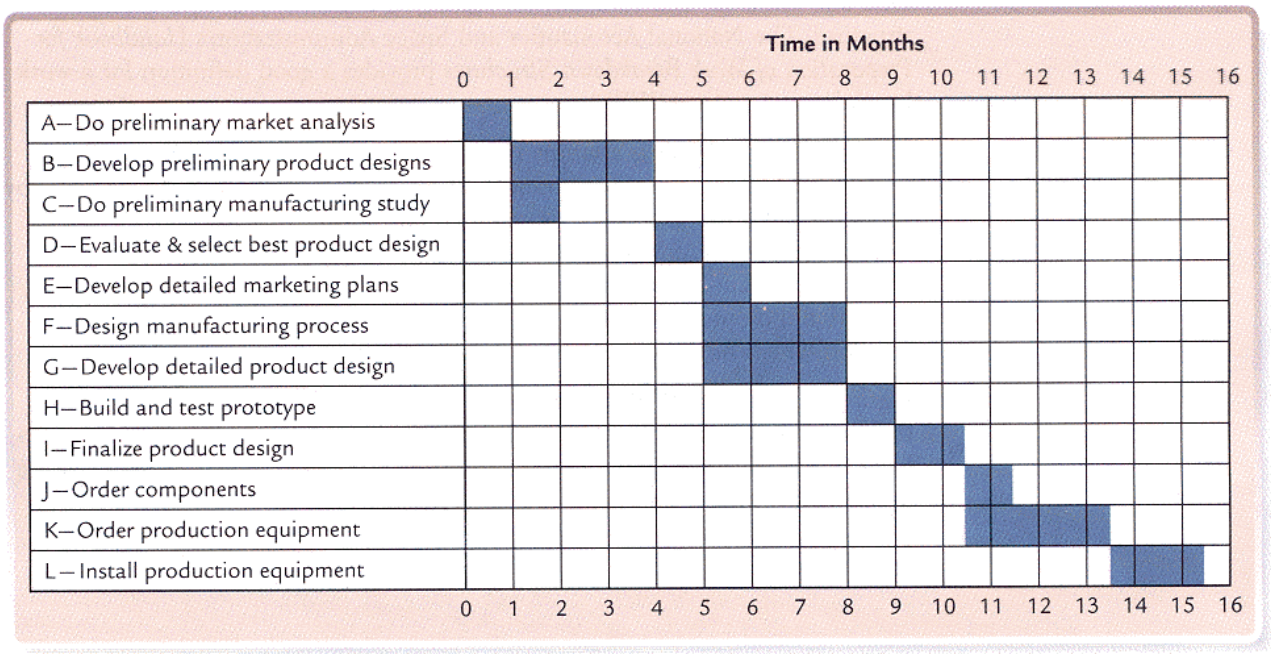
Indice

1. Introduzione

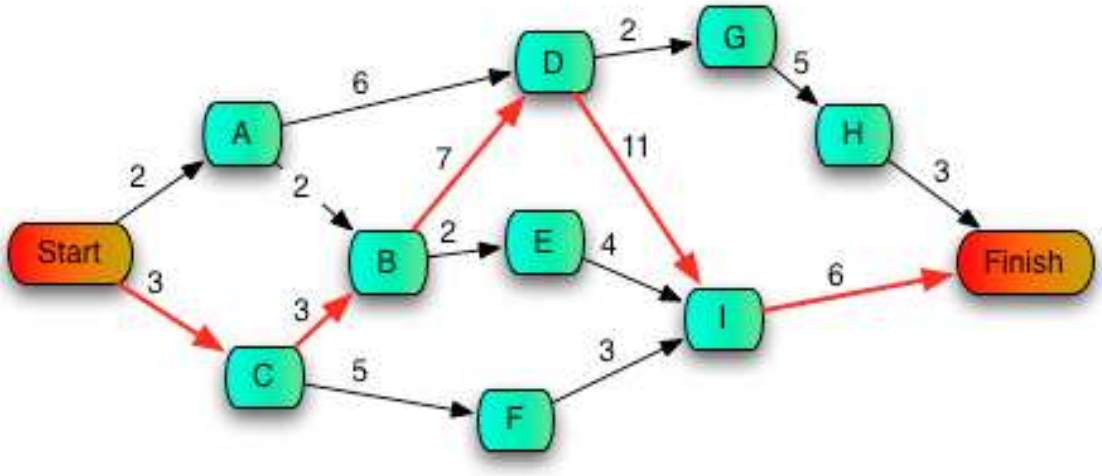
Lo strumento che vogliamo realizzare andrà ad integrare il progetto CASE, un ambiente di sviluppo che assiste i progettisti e i programmatori di un'applicazione in tutte le fasi di progettazione e realizzazione della stessa.
Obiettivo del nostro lavoro è un'applicazione che permetta, attraverso una procedura guidata piuttosto rigida, di rappresentare graficamente i diagrammi di Gantt e Pert, e successivamente di verificare alcune loro proprietà attraverso l'uso di un linguaggio di modellazione e risoluzione di problemi combinatori, Choco Java.

Il diagramma di Gantt è uno strumento di supporto per la gestione dei progetti. E' costruito partendo da un asse orizzontale, suddiviso in intervalli (giorni, settimane o mesi), che rappresenta l'arco temporale del progetto e da un asse verticale per la definizione delle attività che costituiscono il progetto. Barre orizzontali di lunghezza variabile rappresentano la durata di ogni singola attività, alle quali può essere associato un costo. Queste barre possono sovrapporsi durante lo stesso arco temporale, indicando la possibilità di svolgimento parallelo delle attività a cui fanno riferimento. La fine preventivata per il progetto costituisce la "deadline".

Un diagramma di Gantt permette dunque la rappresentazione grafica di un calendario di attività, utile al fine di pianificare, coordinare e tracciare specifiche attività in un progetto, dando una chiara illustrazione dello stato d'avanzamento del progetto rappresentato; di contro, una delle cose non tenute in considerazione in questo tipo di diagrammazione è l'interdipendenza delle attività sottostanti, caratteristica invece del diagramma Pert.



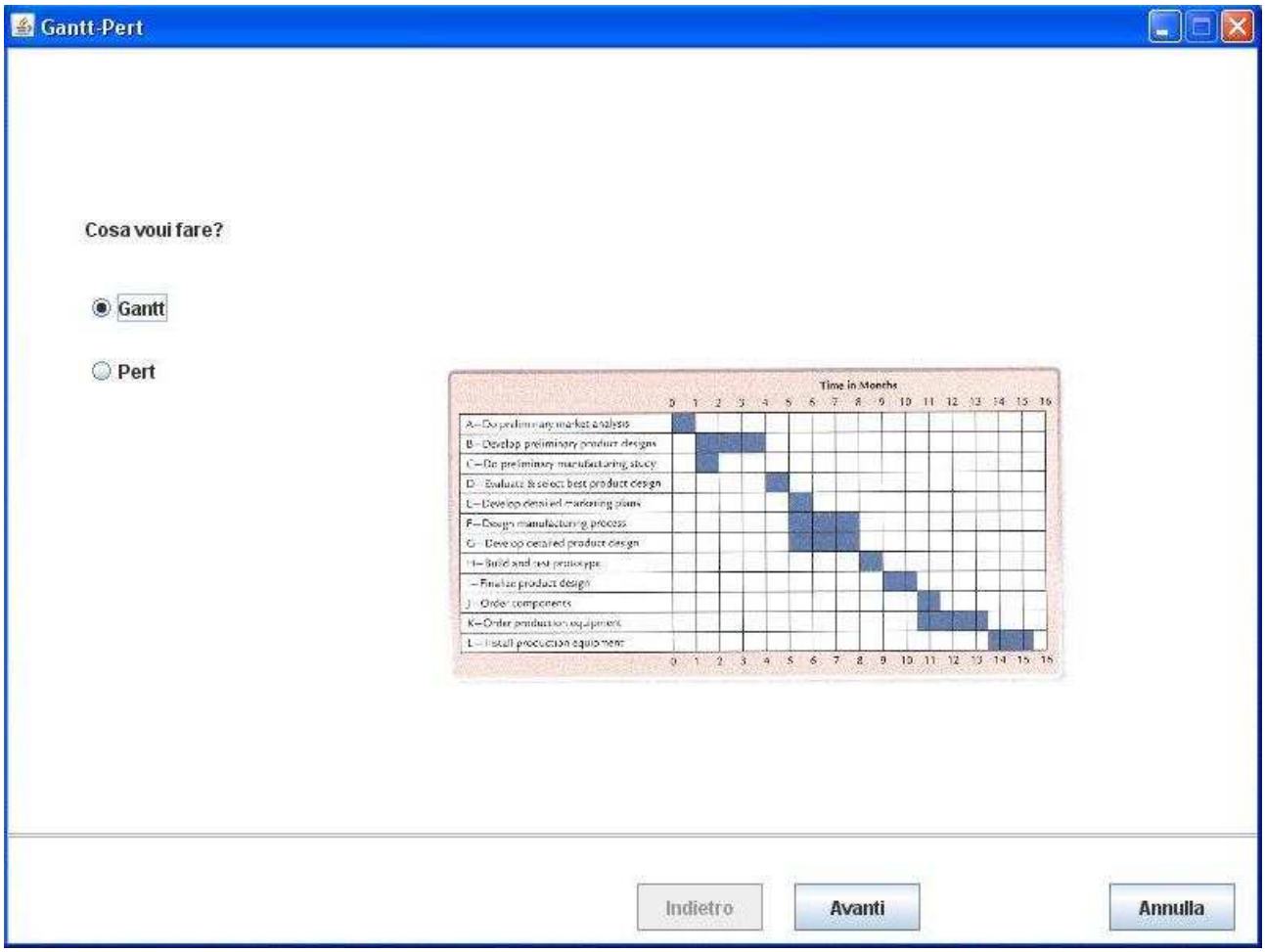
Anche il diagramma Pert è un formalismo grafico di supporto per la gestione dei progetti. Con questa tecnica si tengono sotto controllo le attività di un progetto utilizzando una rappresentazione reticolare che tiene conto della interdipendenza tra tutte le attività necessarie al completamento del progetto.
Un progetto consiste in una serie di attività, rappresentate attraverso nodi, e le dipendenze tra tali attività, raffigurate con degli archi orientati.
Nell'ambito del grafo si individua la sequenza di attività di massima durata ai fini della realizzazione di un progetto. Individuato il percorso critico si tengono sotto stretto controllo le attività che lo compongono, poiché un ritardo di una qualsiasi di queste comporta inevitabilmente un ritardo dell'intero progetto.



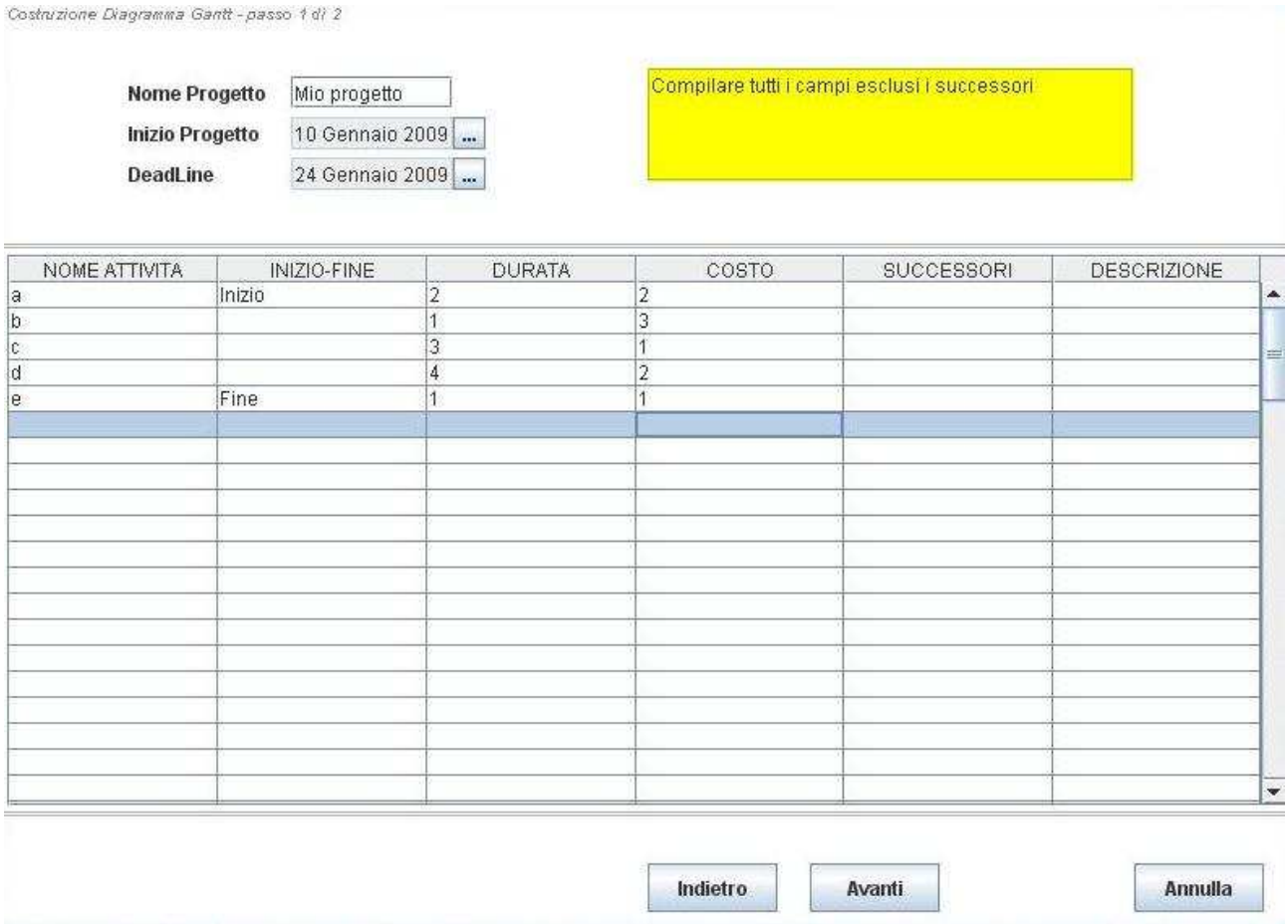
2. Realizzazione interfaccia grafica

L'applicazione è stata realizzata attraverso la libreria grafica Swing di Java e consente la costruzione di diagrammi Gantt o Pert attraverso una sequenza di passi predefinita:

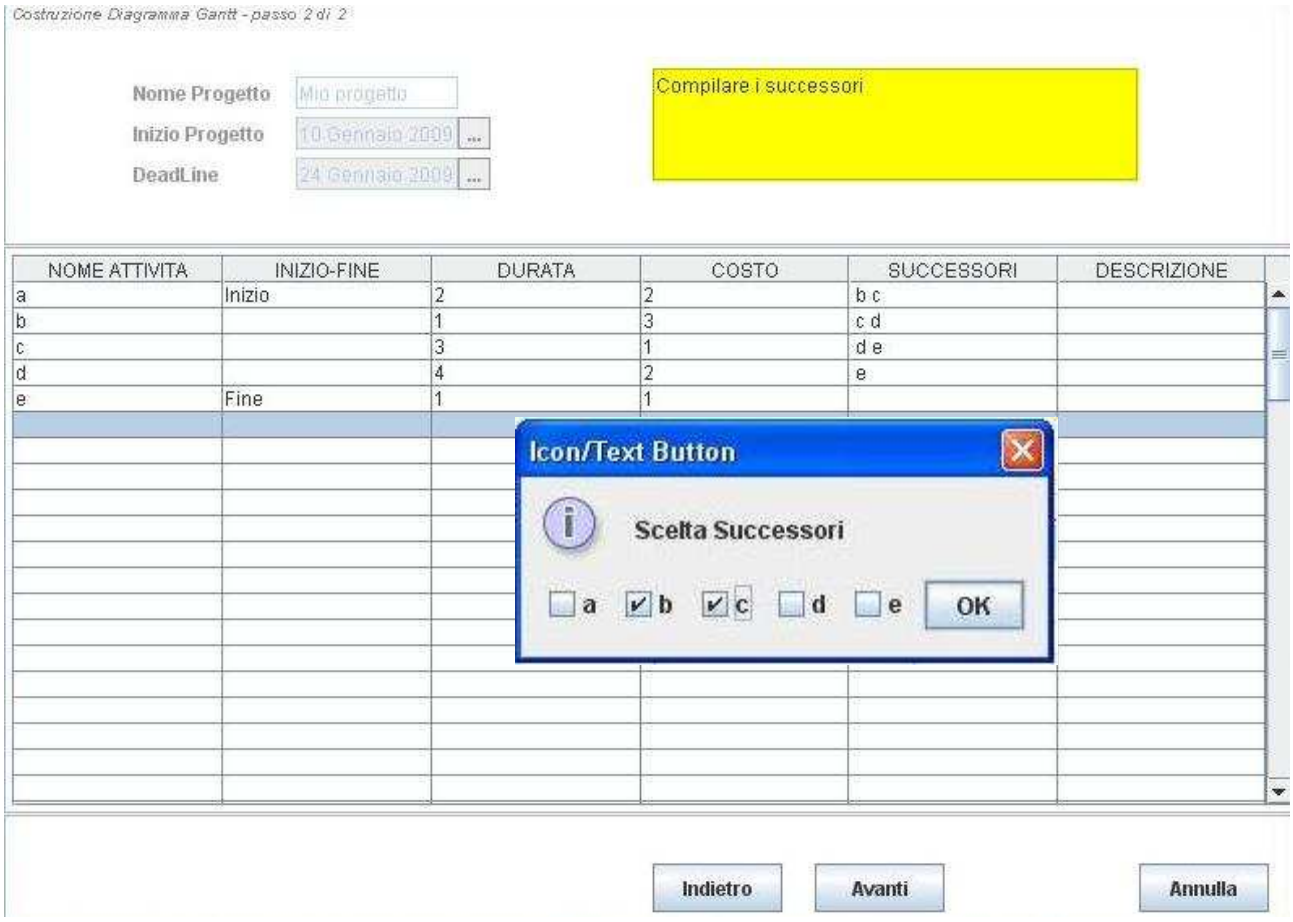
- in un primo momento l'interfaccia permette di selezionare il diagramma da rappresentare.



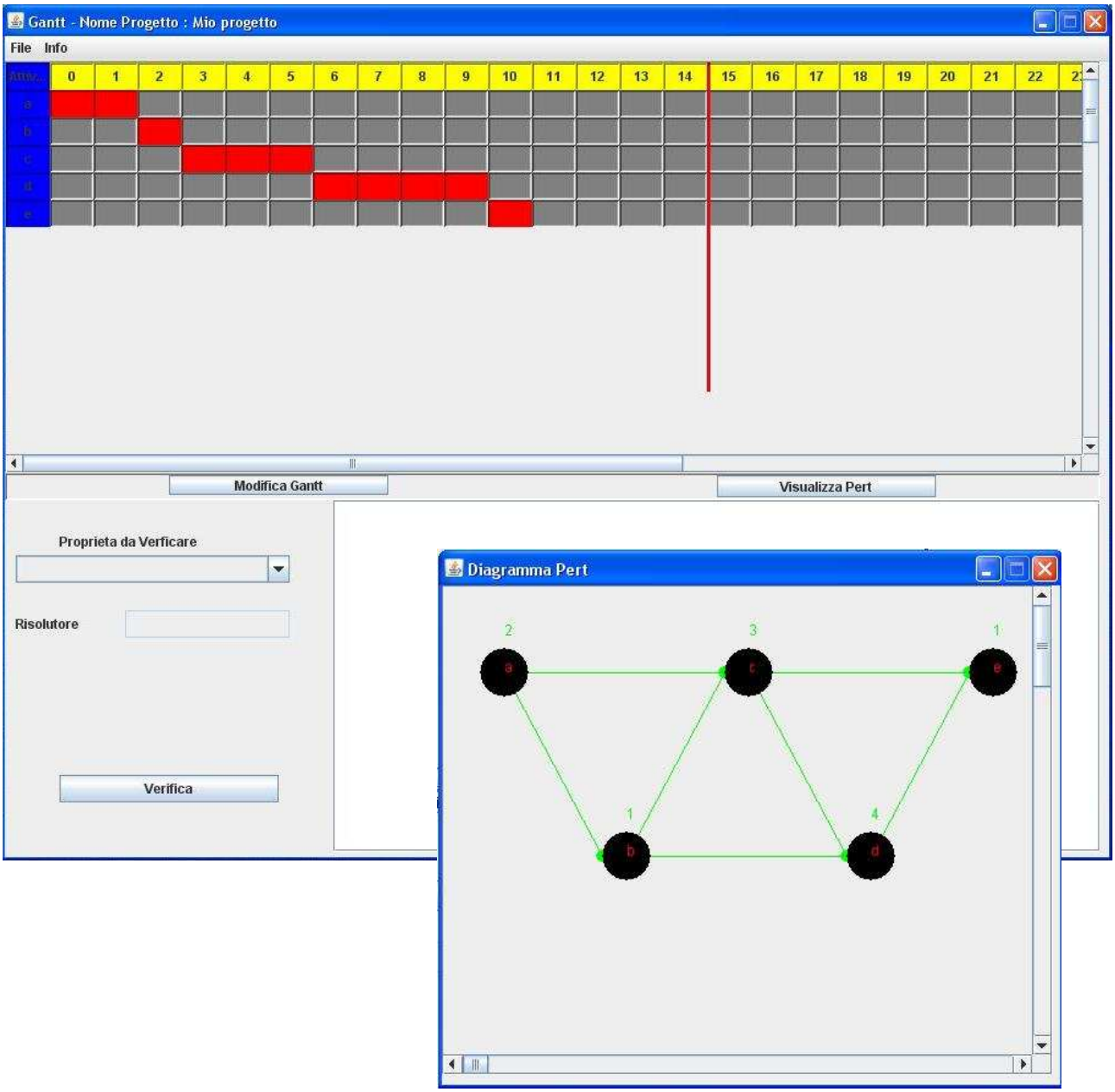
- al passo successivo occorre specificare l'elenco delle attività che fanno parte del progetto, con le relative durate e costi in termini di risorse complessive; informazioni fondamentali sono anche l'attività di inizio progetto e quella di fine progetto. In caso di diagramma di Gantt è necessario inserire la data di inizio progetto e quella della fine preventivata del progetto; quest'ultima rappresenterà la "deadline", cioè l'orizzonte temporale entro il quale tutte le attività dovrebbero essere portate a termine.



- dopo aver inserito correttamente tutte le informazioni necessarie a continuare la costruzione guidata del diagramma, occorre specificare le relazioni di successione tra le attività, esclusa l'attività di fine progetto.

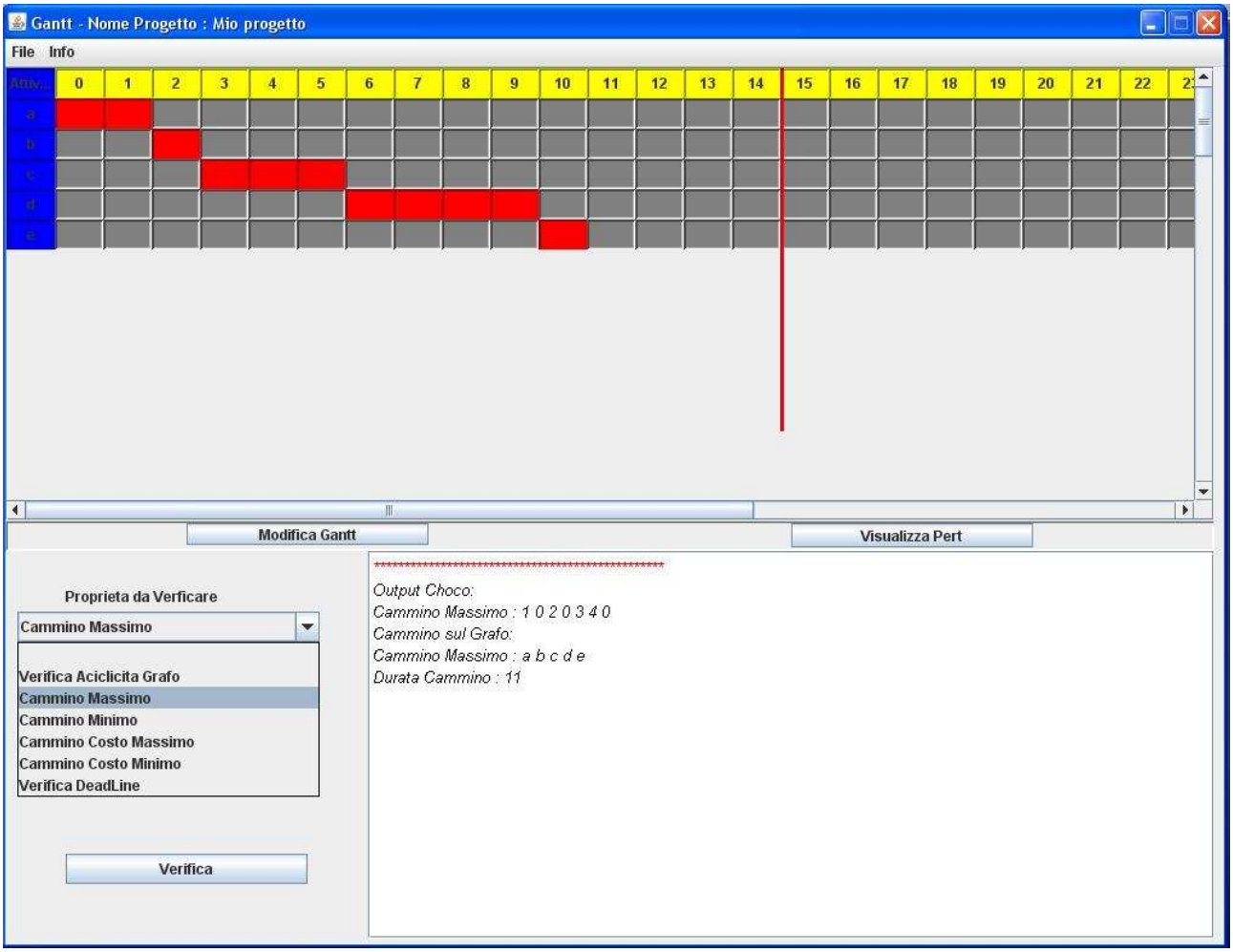


Terminata la costruzione, viene mostrato il diagramma selezionato. L'applicazione permette di tornare ai passi precedenti per modificare le informazioni inserite oppure visualizzare il corrispondente diagramma (Pert o Gantt).



La funzionalità di maggiore interesse è la verifica di alcune proprietà dei diagrammi rappresentati, per la quale verrà mostrato l'output prodotto dall'esecuzione di opportuni risolutori. Proprietà:

- aciclicità del grafo
- cammino di durata massima
- cammino di durata minima
- cammino di costo massimo
- cammino di costo minimo
- rispetto della deadline



3. Programmazione a vincoli: Choco

La modellazione del problema è stata effettuata tramite *Choco*, uno strumento Java per la programmazione a vincoli che ha come obiettivi principali la formulazione e la risoluzione di problemi combinatori. L'idea della programmazione a vincoli è trovare una soluzione al problema che soddisfi certe condizioni e proprietà, espresse nella formulazione del problema stesso.

Un vincolo è una relazione logica tra variabili, ognuna delle quali assume un valore in un determinato dominio; il dominio riduce l'insieme dei possibili valori che le variabili possono ottenere. I vincoli sono caratterizzati da alcune proprietà interessanti:

- possono specificare un'informazione parziale: un vincolo non deve specificare necessariamente i valori delle sue variabili

- sono "non-direzionali", tipicamente un vincolo (ad esempio) su due variabili X e Y può essere utilizzato per dedurre un vincolo su X dato un vincolo sulla Y e viceversa

- sono dichiarativi, cioè precisano quale relazione deve esistere, senza specificare una procedura di calcolo per far rispettare tale relazione

- sono additivi, cioè non è importante l'ordine di imposizione di vincoli, ma il fatto che siano in congiunzione

- sono raramente indipendenti

La programmazione a vincoli permette di risolvere un problema combinatorio modellato tramite un "costraint satisfaction problem" (CSP), ossia una tripla $<X,D,C>$

Variabili: $X = \{X_1,X_2,...,X_n\}$ l'insieme delle variabili del problema

Dominio: D una funzione che associa a ogni variabile i possibili valori che essa può assumere

Vincoli: $C = \{C_1,C_2,...,C_k\}$ l'insieme dei vincoli che limitano il dominio di ogni variabile su cui vengono espressi

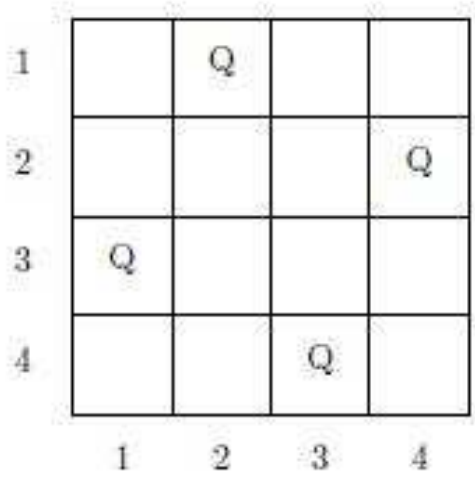
Risolvere un CSP consiste nel trovare una tupla di valori per l'insieme delle variabili, tale da soddisfare tutti i vincoli.

Esistono diversi tipi di variabili:

- Integer variables: variabili definite su domini che contengono valori interi
- Set variables: variabili definite su domini che contengono insiemi di valori
- Real variables: variabili definite su domini continui e che generalmente usano intervalli per rappresentare i valori

Consideriamo un esempio e illustriamone la modellazione in Choco.

Il problema delle N-Regine: dato un intero positivo N, determinare il posizionamento di N regine su una scacchiera NxN tale che non permetta a nessuna regina di colpire nessun'altra regina in una sola mossa. Una regina può muoversi a piacimento, orizzontalmente, verticalmente o diagonalmente.



Il problema delle N-Regine può essere modellato tramite un "costraint satisfaction problem" nel seguente modo:

Variabili: $X = \{X_i \mid i \text{ è un intero in } [1,n]\}$

Dominio: per ogni X_i in X, $D(X_i) = \{j \mid j \text{ è un intero in } [1,n]\}$

Vincoli: l'insieme dei vincoli è dato dall'unione dei seguenti vincoli

- le regine devono essere in linee diverse:

  $C_{lines} = \{X_i \neq X_j \mid i \text{ e } j \text{ sono due interi distinti in } [1,n]\}$

- le regine devono essere in diagonali diverse:

  $C_{diag1} = \{X_i \neq X_j+j-i \mid i \text{ and } j \text{ sono due interi distinti in } [1,n]\}$

  $C_{diag2} = \{X_i \neq X_j+i-j \mid i \text{ and } j \text{ sono due interi distinti in } [1,n]\}$

Elemento centrale di un programma Choco è l'oggetto della classe *Problem*, attraverso il quale è possibile creare le variabili e definire i vincoli:

```
Problem myPb = new Problem();
```

Esistono tre tipi di variabili:

IntDomainVar : definisce domini discreti dove i valori sono interi
RealVar : definisce domini continui e usa intervalli per rappresentare i valori
SetVar : definisce un insieme di domini discreti dove il valore di una variabile è un insieme

Una volta creato il problema, le variabili vengono create usando i metodi della classe *Problem* invece che il tradizionale costruttore Java. Questo permette di assicurare che i vincoli e i tipi delle variabili restino compatibili. Per esempio, creare una variabile con dominio finito e discreto richiede il seguente codice:

```
IntDomainVar v1 = myPb.makeEnumIntVar("var1", 1, 10);
```

Le informazioni sulle variabili sono accessibili e modificabili attraverso i principali metodi pubblici della classe di appartenenza: *getInf(), getSup(), getVal(), isInstantiated(), getDomainSize(), setInf(), setSup(), setVal(), setDomain(), ...*

I vincoli sono definiti su un problema usando il metodo *post* sull'oggetto *Problem* : post(Costraint c).Per esempio, aggiungere un vincolo di disuguaglianza tra due variabili si scrive così:

myPb.post(myPb.neq(vars1, vars2));

In base al tipo delle variabili considerate si hanno a disposizione diversi tipi di vincoli; sulle variabili intere i vincoli possono essere:

- aritmetici : uguaglianza, differenza, confronto, combinazione linare, operatori aritmetici

neq(IntExp v1, IntExp v2) : v1 != v2
eq(IntExp v1, IntExp v2) : v1 = v2
leq(IntExp v1, IntExp v2) : v1 <= v2
lt(IntExp v1, IntExp v2) : v1 < v2
minus(IntExp exp1, IntExp exp2) : exp1 - exp2
plus(IntExp exp1, IntExp exp2) : exp1 + exp2
times(IntVar x, IntVar y, IntVar z) : z = x * y
mult(int coef, IntExp exp) : coef * exp
scalar(int[] coef, IntVar[] v): coef[1]*v[1]+..+coef[n]*v[n]
sum(IntVar[] vars): vars[1] + ... + vars[n]
abs(IntVar x, IntVar y): x = |y|
min(IntVar x, IntVar y, IntVar z): z = min(x,y)
max(IntVar x, IntVar y, IntVar z): z = max(x,y)
min(IntVar[] vs, IntVar z): z = min({vs[1],...,vs[n]})
max(IntVar[] vs, IntVar z): z = min({vs[1],...,vs[n]})

- booleani

  or(Constraint c1, Constraint c2)
  and(Constraint c1, Constraint c2)
  implies(Constraint c1, Constraint c2)
  not(Contraint c)

- vincoli di channelling

  inverseChanneling(IntVar[] x, IntVar[] y): x[i]=j <=> y[j]=i

  booleanChanneling(IntVar bv, IntVar x, int j): *bv* è una variabile booleana con dominio {0,1} e *x* una variabile intera. Questo vincolo assicura x= j <=> bv = 1, agendo da "osservatore" del valore j

- vincoli binari definiti dall'utente

- vincoli globali: permettono di filtrare efficientemente alcuni valori inconsistenti introducendo regole di filtraggio aggiuntive

  pb.allDifferent(IntVar[] vars): assicura che tutti gli elementi della variabile abbiano valori differenti

  pb.occurrence(IntVar[] vars, int v, IntVar occurrence): assicura che *occurrence* venga instanziata al numero di occorrenze di *v* nella lista di variabili *vars*

pb.globalCardinality(IntVar[] vars, int[] low, int[] up): assicura che il numero di occorrenze del valore 1 in tutte le variabili *vars* sia tra *low[0]* e *up[0]* e in generale che il numero di occorrenze del valore i in *vars* sia tra *low[i-1]* e *up[i-1]*

pb.lex(IntVar[] x, IntVar[] y) : definisce un ordine lessicografico su due vettori di variabili intere

$x = \langle x\_0, ..., x\_n \rangle$ e $y = \langle y\_0, ..., y\_n \rangle : x <_{lex} y$

pb.atMostNValue(IntVar[] vars, IntVar nvalue) : forza il numero di differenti valori che occorrono in *vars* ad essere al più *nvalue*

Vincoli sulle variabili di tipo *SetVariables* :

- vincoli sugli insiemi

member(SetVar sv1, int val): specifica che la variabile *sv1* contiene il valore *val*

notMember(SetVar sv1, int val): specifica che la variabile *sv1* non contiene il valore *val*

setDisjoint(SetVar sv1, SetVar sv2): specifica che le variabili *sv1* e *sv2* sono insiemi disgiunti

setInter(SetVar sv1, SetVar sv2, SetVar inter): specifica che l'insieme *inter* è l'intersezione degli insiemi *sv1* e *sv2*

eqCard(SetVar sv, int val): vincola la cardinalità dell'insieme *sv* ad essere pari al valore *val*

geqCard(SetVar sv, int val): vincola la cardinalità dell'insieme *sv* ad essere maggiore o uguale al valore *val*

leqCard(SetVar sv, int val): vincola la cardinalità dell'insieme *sv* ad essere minore o uguale al valore *val*

- vincoli sugli insiemi e variabili intere: vincoli misti

member(SetVar sv1, IntVar var)
notMember(SetVar sv1, IntVar var)
eqCard(SetVar sv, IntVar iv)
geqCard(SetVar sv, IntVar iv)
leqCard(SetVar sv, IntVar iv)

Il problema è l'elemento centrale di un modello Choco in quanto permette la creazione delle variabili e dei vincoli; ma il controllo del processo di ricerca, senza utilizzare tool predefiniti, è affidato alla classe Solver. Per ottenere le soluzioni del modello definito occorre invocare il metodo *solve().*

```
Solver s = p.getSolver();
p.solve(true);
int soluzioniTrovate = s.getSearchSolver().solutions.size();
if(soluzioniTrovate>0) {
        String[] soluzioni = new String[n+1];
        Solution sol = (Solution) s.getSearchSolver().solutions.get(0);
        for (int j = 0; j < n+1; j++)
                soluzioni[j] = ""+sol.getValue(j);
        return soluzioni;
}
 else{ return null;
}
```

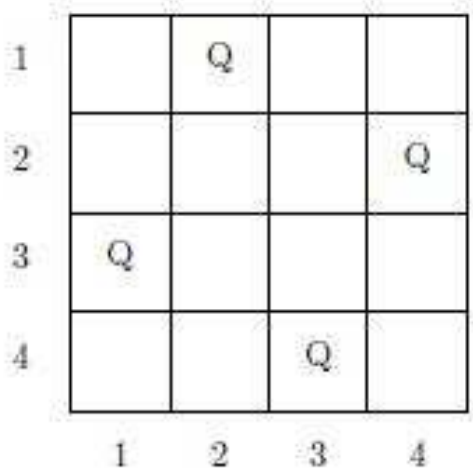
Choco permette di ottimizzare il valore di una variabile attraverso la definizione di una funzione obiettivo espressa come vincolo sulla variabile e il resto del problema:

minimize(IntVar obj, boolean restart)
maximize(IntVar obj, boolean restart)

Se il parametro *restart* è *true* il solutore ricomincia la ricerca dopo aver trovato una soluzione, altrimenti fa backtracking a partire dall'ultima soluzione trovata.

Per concludere la trattazione su Choco riportiamo la traduzione del problema delle N-Regine:

```
//1- Creare il problema
 Problem pb = new Problem();
//2- Creare le variabili
 IntVar[] queens = new IntVar[n];
for (int i = 0; i < n; i++) {
        queens[i] = pb.makeEnumIntVar("Q" + i, 1, n); }
//3- Definire i vincoli
for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
                int k = j – i;
                pb.post(pb.neq(queens[i],queens[j]));
        pb.post(pb.neq(queens[i],pb.plus(queens[j],k)));
        pb.post(pb.neq(queens[i], pb.minus(queens[j], k))); } }
//4- Ricerca di tutte le possibili soluzioni
pb.solveAll();
//5- Stampa delle soluzioni
System.out.println("NbSol: " + pb.getSolver().getNbSolutions());
```



4. Traduzione in Choco

Il punto di partenza della modellazione del nostro problema è il diagramma Pert, è noto che un diagramma di Gantt è sempre riconducibile a un diagramma Pert. Nella fase di realizzazione dell'interfaccia grafica e di implementazione degli algoritmi che ne regolano il funzionamento, sono state utilizzate delle strutture dati per la rappresentazione dei diagrammi; queste strutture dati diverranno l'input per la fase di modellazione e la successiva analisi formale. In particolare, abbiamo:

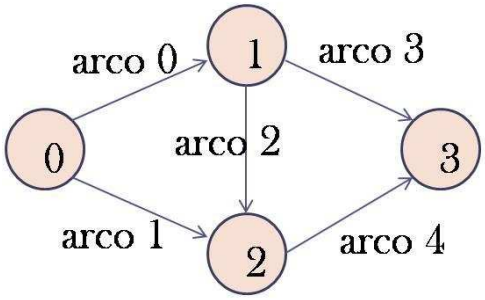
- la lista delle attività
- la matrice delle adiacenze delle attività
- il vettore delle durate
- il vettore dei costi

Per definire il nostro problema facciamo uso di alcuni vettori:

- una matrice *grafo* con tante righe quanti sono gli archi nel grafo e due colonne: l'informazione nella prima colonna rappresenta il nodo di partenza di un arco, mentre nella seconda si trova il nodo di arrivo.

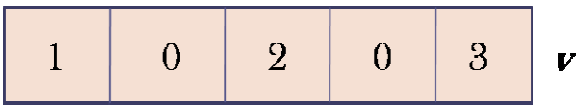
0	1	arco 0
0	2	arco 1
1	2	arco 2
1	3	arco 3
2	3	arco 4

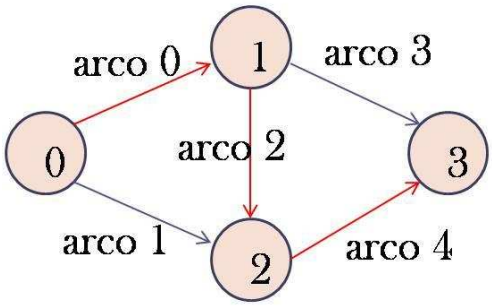
grafo



- due array di interi, uno per memorizzare i valori delle durate delle singole attività e l'altro per i costi

- un vettore *v* in cui memorizziamo, nell'i-sima posizione, l'informazione circa l'appartenenza dell'i-simo arco al cammino obiettivo della ricerca (cammino di durata massima o minima, cammino costo minimo o massimo). Il dominio di *v* sono valori interi in [0,n], dove ogni singolo valore indica l'ordine di attraversamento dell'arco corrispondente nel cammino trovato






```
IntDomainVar[] v = new IntDomainVar[n];
   for (int i=0; i<num_archi;i++) {
         v[i] = problem.makeEnumIntVat("arco"+i,0,n);
   }
```

Grazie a queste strutture dati abbiamo una corrispondenza biunivoca tra ogni elemento del vettore *v* e ogni riga della matrice *grafo.*

L'obiettivo del nostro studio è l'analisi di alcune proprietà dei diagrammi trattati, che possono essere verificate attraverso Choco:

- cammino di durata massima
- cammino di durata minima
- cammino di costo massimo
- cammino di costo minimo
- rispetto della deadline

Per il calcolo delle durate e dei cammini, massimi o minimi, utilizziamo una funzione di ottimizzazione che viene espressa come un vincolo sulla variabile di interesse:

```
Problem p = new Problem();
IntDomainVar durata = p.makeEnumIntVar("durata",0,100000);
p.maximize(durata,false);
//oppure
p.minimize(durata,false);
IntDomainVar costo = p.makeEnumIntVar("costo",0,100000);
p.maximize(costo,false);
//oppure
p.minimize(costo,false);
```

Per la ricerca del cammino da ottimizzare tramite la funzione obbiettivo descritta in precedenza, definiamo una serie di vincoli:

1) corrispondenza tra la durata di un arco e il vettore *v* degli archi: dato un array di booleani, se nell'i-sima posizione c'è il valore 1, la durata relativa a quell'arco va considerata nel calcolo finale perchè significa che il corrispondente arco nel vettore v ha valore > 0, cioè è stato attraversato

```
IntDomainVar[] bool = new IntDomainVar[n];
for(int i = 0; i < n; i++){
        bool[i] = p.makeEnumIntVar("bool",0,1);
        p.post(p.implies(p.eq(0,v[i]),p.eq(bool[i],0)));
        p.post(p.implies(p.eq(bool[i],0),p.eq(v[i],0)));
}
```

Si usa il vettore booleano *bool* in cui il valore i-simo indica se la corrispondente cella (i-sima) del vettore *v* contiene un valore > 0, cioè se l'arco appartiene al cammino. L'assegnazione dei valori del vettore booleano è legata dalla doppia implicazione al vettore *v*:

p.post(p.implies(p.eq(0,v[i]),p.eq(bool[i],0))); v[i]=0 -> bool[i]=0
p.post(p.implies(p.eq(bool[i],0),p.eq(v[i],0))); bool[i]=0 -> v[i]=0

E' stata adottata questa soluzione poiché Choco non permette di utilizzare la valutazione di un espressione all'interno di funzioni aritmetiche, cioè non è possibile calcolare

$$\sum_{i=0}^{n} (\text{p.lt(0,v[i])}) * \text{durata[i]}$$

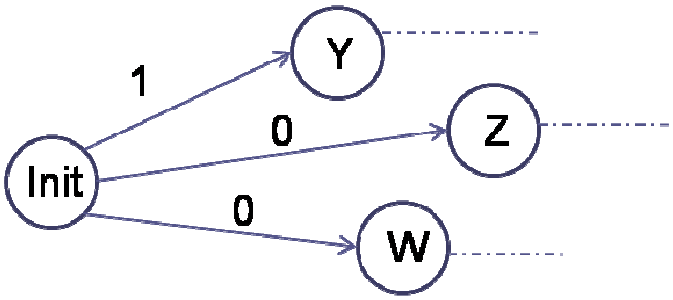
2) corrispondenza tra un arco e la sua durata: dato l'array booleano *bool* e il vettore delle durate leghiamo i due vettori alla variabile oggetto della funzione obiettivo. Il seguente vincolo forza il valore della variabile *durata* ad essere uguale al prodotto scalare tra il vettore delle durate degli archi e il vettore booleano:

p.post(p.eq(p.scalar(durateArchi,bool),durata));

3) vincolo sul nodo di partenza: solamente un arco, tra quelli uscenti, può essere presente nel cammino.

IntDomainVar[] archiIniziali = archiIniziali(v,grafo);

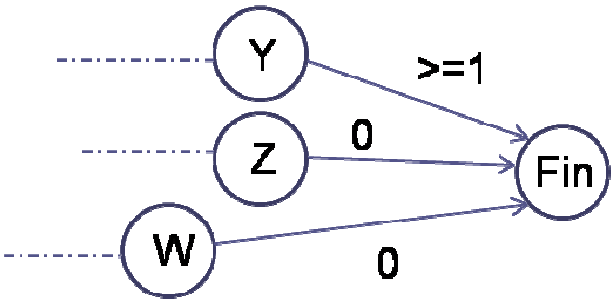
p.post(p.eq(p.sum(archiIniziali),1));



Choco non offre la possibilità di esprimere il quantificatore universale; per i nostri scopi è stato utilizzato l'operatore d'aggregazione somma nella maniera più opportuna.

4) vincolo sul nodo di arrivo: solamente un arco, tra quelli entranti, può essere presente nel cammino; la somma degli "archi finali" deve essere >= 1 , cioè se esiste una soluzione dobbiamo arrivare al nodo finale

IntDomainVar[] archiFinali = archiFinale(v,grafo,matriceAd);
p.post(p.leq(1,p.sum(archiFinali)));



Il vincolo maggiore uguale a uno è dovuto al fatto che può esistere un grafo cha abbia un solo arco, uscente dal nodo iniziale ed entrante nel nodo di fine progetto; per come sono stati espressi gli altri vincoli la sua numerazione sarà proprio pari al valore 1.

Grazie alla congiunzione di tutti i vincoli espressi, si avrà che solamente un arco entrante nel nodo finale avrà valore diverso da zero come mostrato nella figura.

Le funzioni *archiInizali* e *archiFinale* servono ad ottenere, rispettivamente, l'insieme degli archi uscenti dal nodo iniziale e quelli entranti nel nodo finale:

archiIniziali(IntDomainVar[] v,int[][] grafo): restituisce un array di puntatori che rappresenta un sottoinsieme del vettore *v*

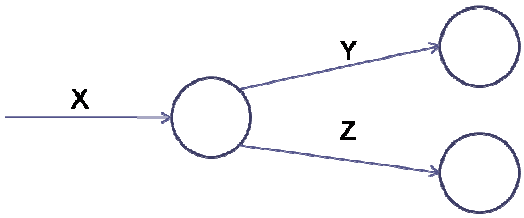
archiFinale(IntDomainVar[] v,int[][] grafo): restituisce un array di puntatori che rappresenta un sottoinsieme del vettore *v*

5) vincolo sulla numerazione di un arco e dei suoi successori: se un arco ha valore *i* maggiore di zero, la somma dei suoi successori nel cammino è uguale a *i+1*

```
for(int i = 0; i < n ; i ++){
        IntDomainVar[] succ = successori(v,i,grafo);
        if(succ!=null){
p.post(p.implies(p.lt(0,v[i]),p.eq(p.sum(succ),p.plus(v[i],1))));
        }
}
```

6) vincolo sulla numerazione di un arco e dei suoi predecessori: se un arco ha valore i maggiore di zero, la somma dei suoi predecessori nel cammino è uguale a i-1

```
for(int i = 0; i < n ; i ++){
    IntDomainVar[] pred = predecessori(v,i,grafo);
    if(pred!=null){
     p.post(p.implies(p.lt(0,v[i]),p.eq(p.sum(pred),p.minus(v[i],1))));
    }
}
```



p.post(p.implies(p.lt(0,v[i]),p.eq(p.sum(succ),p.plus(v[i],1))))

X +1 = Y +Z

p.post(p.implies(p.lt(0,v[i]),p.eq(p.sum(pred),p.minus(v[i],1))));

Z = X -1

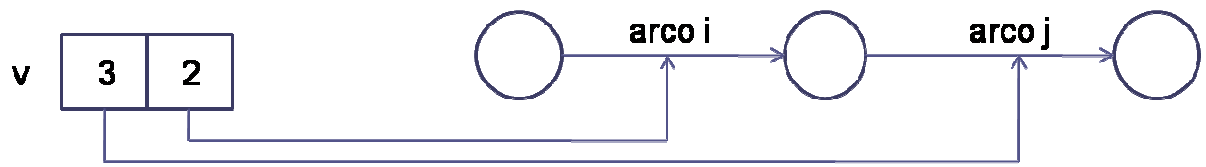
Y = X -1

La soluzione a questo sistema implica che
Y >0 -> Z = 0 and Z >0 -> Y =0 (X ha valore assegnato da Choco)

7) Per ogni coppia di archi i j :
se j è successore di i nel grafo → v[i] =0 or v[j] = 0 or v[j] = v[i]+1

```
for(int i = 0; i < n ;i++){
        for(int j = 0; j <n ; j++){
                if(grafo[i][1] == grafo[j][0]){
        p.post(p.or(p.eq(v[j],0),p.eq(v[i],0),p.eq(v[j],p.plus(v[i],1))));
                }
        }

}
```



Negli altri due casi (v[i] =0 or v[j] = 0), ancora una volta grazie alla congiunzione di tutti i vincoli espressi, è implicato che se v[i]=0 allora v[j]=0 e viceversa.

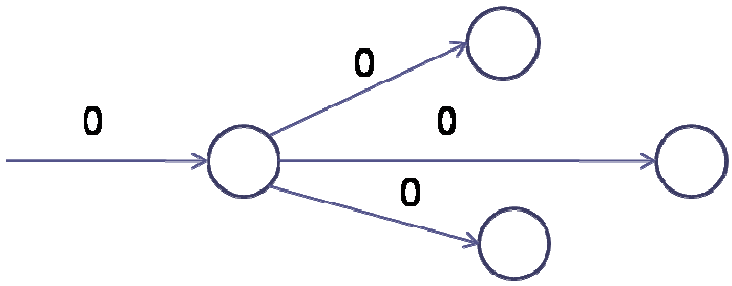
8) vincolo sui successori di un arco: se nessuno dei successori di un arco è presente nel cammino allora l'arco stesso non è presente nel cammino

```
        for(int i = 0; i < n ; i ++){
                IntDomainVar[] succ = successori(v,i,grafo);
                if(succ!=null){
                        p.post(p.implies(p.eq(p.sum(succ),0),p.eq(v[i],0)));
                }
        }
```

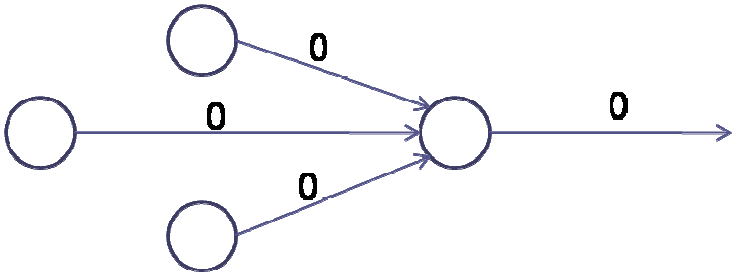
9) vincolo sui predecessori di un arco: se nessuno dei predecessori di un arco è presente nel cammino allora l'arco stesso non è presente nel cammino

```
for(int i = 0; i < n ; i ++){
        IntDomainVar[] pred = predecessori(v,i,grafo);
        if(pred!=null){
                p.post(p.implies(p.eq(p.sum(pred),0),p.eq(v[i],0)));
        }
}
```

```
..
p.post(p.implies(p.eq(p.sum(succ),0),p.eq(v[i],0)));
..
```



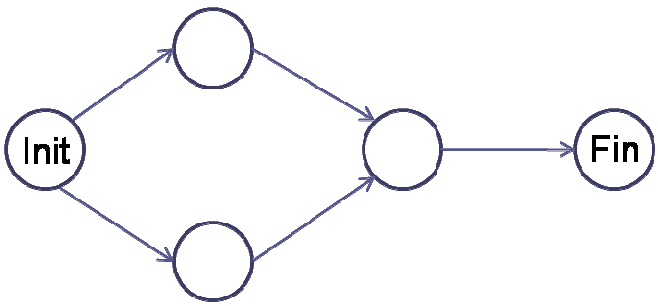

```
..
p.post(p.implies(p.eq(p.sum(pred),0),p.eq(v[i],0)));
..
```

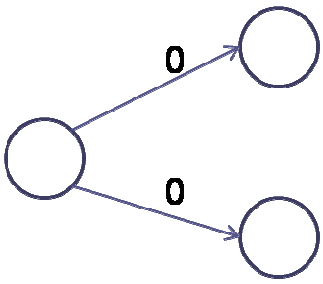


10) Per ogni arco *i*, esclusi quelli iniziali, se non esistono predecessori allora il valore di v[i] è 0 e tutti i successori hanno valore pari a 0. Questo vincolo è stato introdotto per evitare problemi nell'analisi di grafi spezzati come quello in figura

```
for(int i = archiIniziali.length; i < n ; i ++){
        IntDomainVar[] pred = predecessori(v,i,grafo);
        if(pred!=null){

        p.post(p.implies(p.lt(0,v[i]),p.eq(p.sum(pred),p.minus(v[i],1))));
        }else{
        try{
        IntDomainVar[] succ = successori(v,i,grafo);
        v[i].setVal(0);
        if(succ!=null)
                p.post(p.eq(p.sum(succ),0));
        }catch(Exception ex){}
        }
}
```

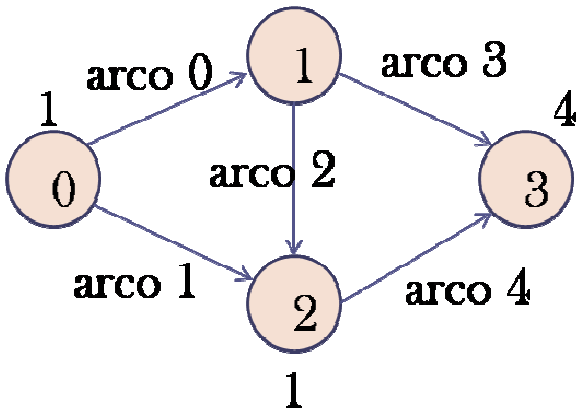




Tutti i vincoli descritti consentono di trovare il cammino massimo o minimo nel grafo che diamo in input a Choco. I vincoli sui successori e predecessori rendono inutile il vincolo *All-Different* , il quale specifica che ogni valore degli elementi del vettore *v* è diverso da ogni altro.
Vediamo un esempio di soluzione non ammissibile:

0	1	arco 0
0	2	arco 1
1	2	arco 2
1	3	arco 3
2	3	arco 4

grafo



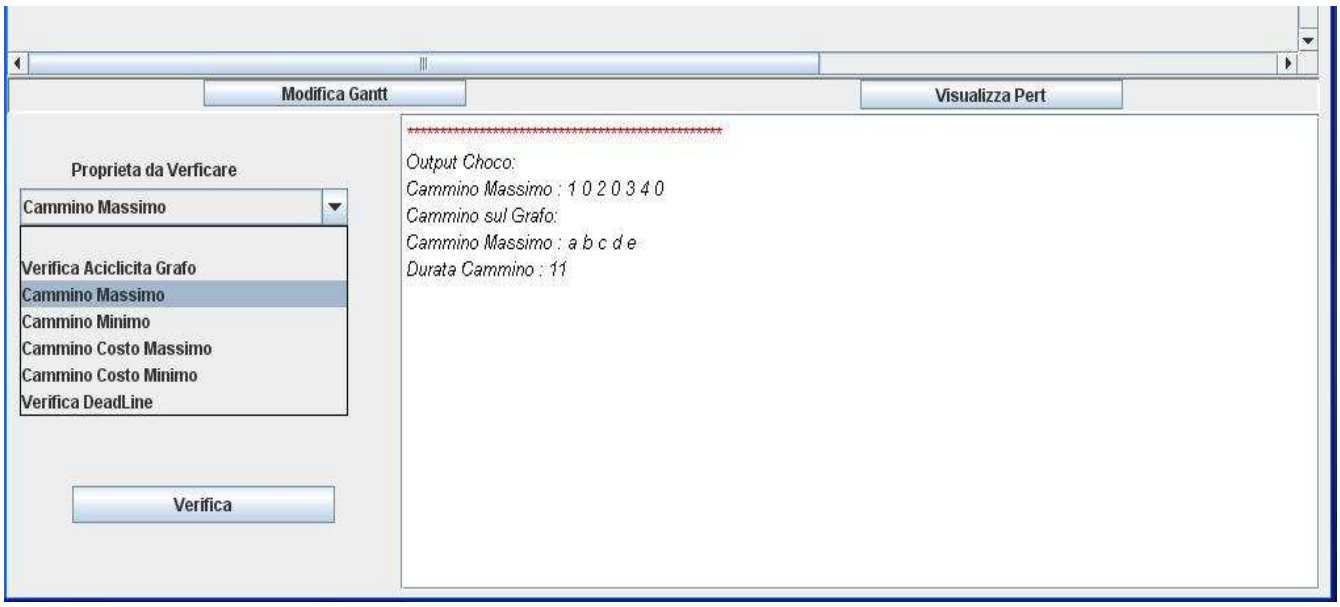
1	1	2	0	3	*v*
arco 0	arco 1	arco 2	arco 3	arco 4	

Il vettore v non rappresenta una soluzione valida per il problema perché *arco1* non può essere pari a 1, poiché la somma degli archi iniziali deve essere pari a 1. Inoltre il suo successore dovrebbe essere pari a 4 ed invece è pari a 3. In definitiva, il vettore v, come rappresentato in figura, non è una soluzione in quanto non rispetta i vincoli specificati per il problema.

Una volta descritti i vincoli, risolviamo il problema invocando il solutore; ci facciamo restituire tutte le eventuali soluzioni trovate (il *true* come parametro del metodo *solve(),* indica che il solutore non si deve fermare alla prima soluzione ma le deve trovare tutte)

```
Solver s = p.getSolver();
p.solve(true);
int soluzioniTrovate = s.getSearchSolver().solutions.size();
if(soluzioniTrovate>0){
        String[] soluzioni = new String[n+1];
        Solution solution =
                        (Solution) s.getSearchSolver().solutions.get(0);
        for (int j = 0; j < n+1; j++)
                soluzioni[j] = ""+solution.getValue(j);
        return soluzioni;
}
else return null;
```

Il risultato di tale ricerca verrà visualizzato tramite un'apposita area dell'interfaccia grafica



5. Verifica dell'aciclicità con NuSMV

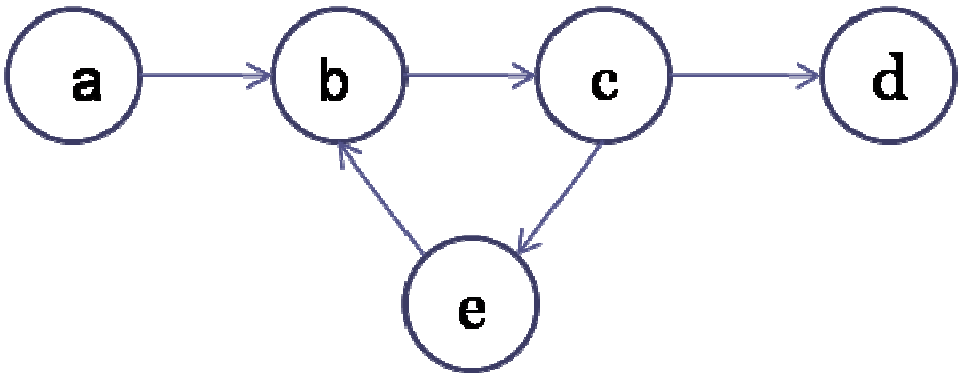
Un'altra proprietà importante che vogliamo verificare sui nostri diagrammi è l'assenza di cicli.
A questo scopo generiamo un file .smv che modella il nostro diagramma sfruttando le strutture dati a disposizione: matrice di adiacenza e lista delle attività.
Attraverso l'uso del risolutore NuSMV saremo in grado di determinare l'esistenza o meno di un cammino ciclico, attraverso la seguente specifica LTL :

((stato = "attività x" & evento = "arco xy")) -> X (G stato != x))

Se l'output del risolutore è *true* non esiste un ciclo, altrimenti viene mostrato un controesempio che lo rappresenta

Mostriamo un esempio:

Ciclo: b-c-e



Codifica NuSMV:

1. Definizioni delle variabili stato ed evento.
   public static String getEvento(boolean[][][] m,String[] attivita);
   public static String getStato(String[] attivita);
2. Definizione della sezione TRANS
   public static String getTrans(boolean[][][] m,String[] att);
3. Definizione della sezione Main per l'attivita specifica
   public static String getMain(boolean[][][] m,String[] att,int at);
4. Funzione che restitisce il file .smv codifica del grafo
   public static String getNuSMV(boolean[][][] m, String[] att, int a);

Codice NuSMV per l'attivita b (un file .smv per ogni attività):

```
MODULE diagramma
VAR
stato : { a,e,b,c,d};
evento : { ab,eb,bc,ce,cd,null};
TRANS
case
stato = a &  evento = ab : next(stato) = b;
stato = e &  evento = eb : next(stato) = b;
stato = b &  evento = bc : next(stato) = c;
stato = c &  evento = ce : next(stato) = e;
stato = c &  evento = cd : next(stato) = d;
1 : next(stato) = stato;
esac
MODULE main
VAR
diagra : diagramma;
ASSIGN
init(diagra.stato):=b;
LTLSPEC
(  diagra.stato = b & (  diagra.evento = bc   ) -> X ( G diagra.stato !=b ) )
```

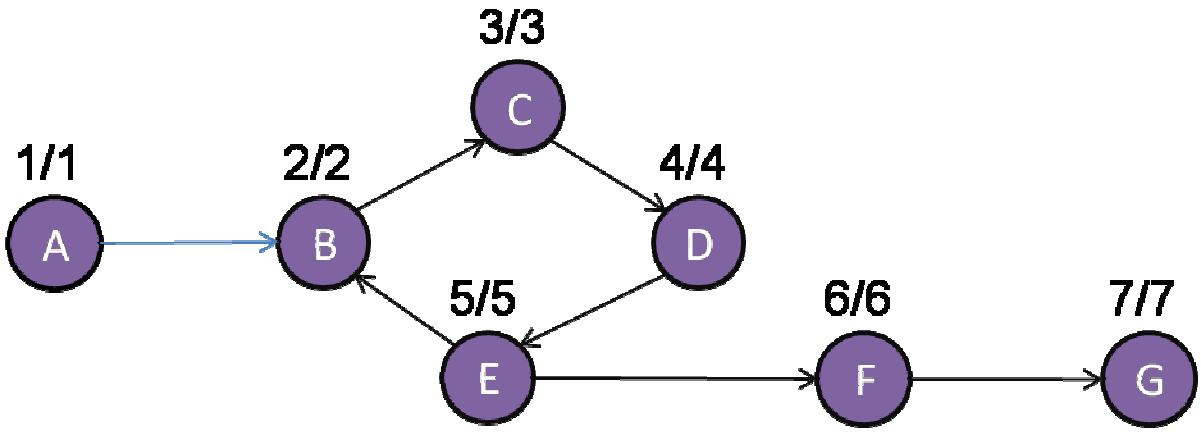
Output NuSMV per l'attivita b:

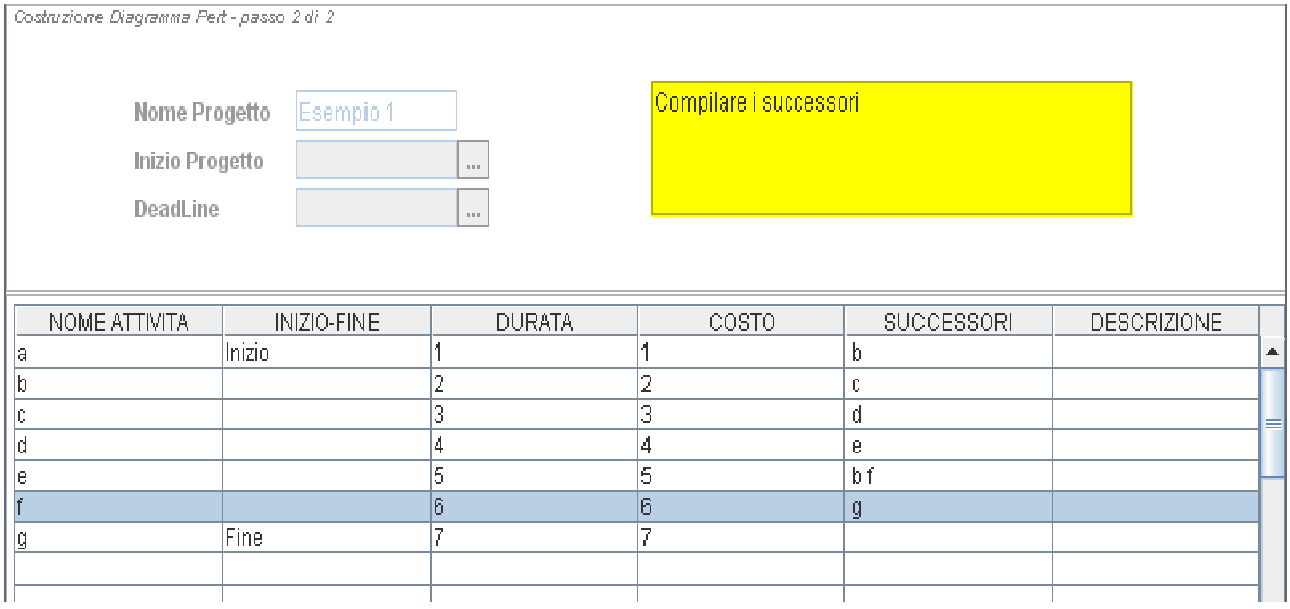
```
-- specification ((diagra.stato = b & diagra.evento = bc) ->  X ( G diagra.stato
!= b))  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-- Loop starts here
-> State: 1.1 <-
  diagra.stato = b
  diagra.evento = bc
-> Input: 1.2 <-
-> State: 1.2 <-
  diagra.stato = c
  diagra.evento = ce
-> Input: 1.3 <-
-> State: 1.3 <-
  diagra.stato = e
  diagra.evento = eb
-> Input: 1.4 <-
-> State: 1.4 <-
  diagra.stato = b
  diagra.evento = bc
```

6. Esempi

Riportiamo alcuni esempi per mostrare il funzionamento e la correttezza dell'applicazione realizzata.
Ogni nodo è etichettato con il nome dell'attività e con la coppia durata/costo.

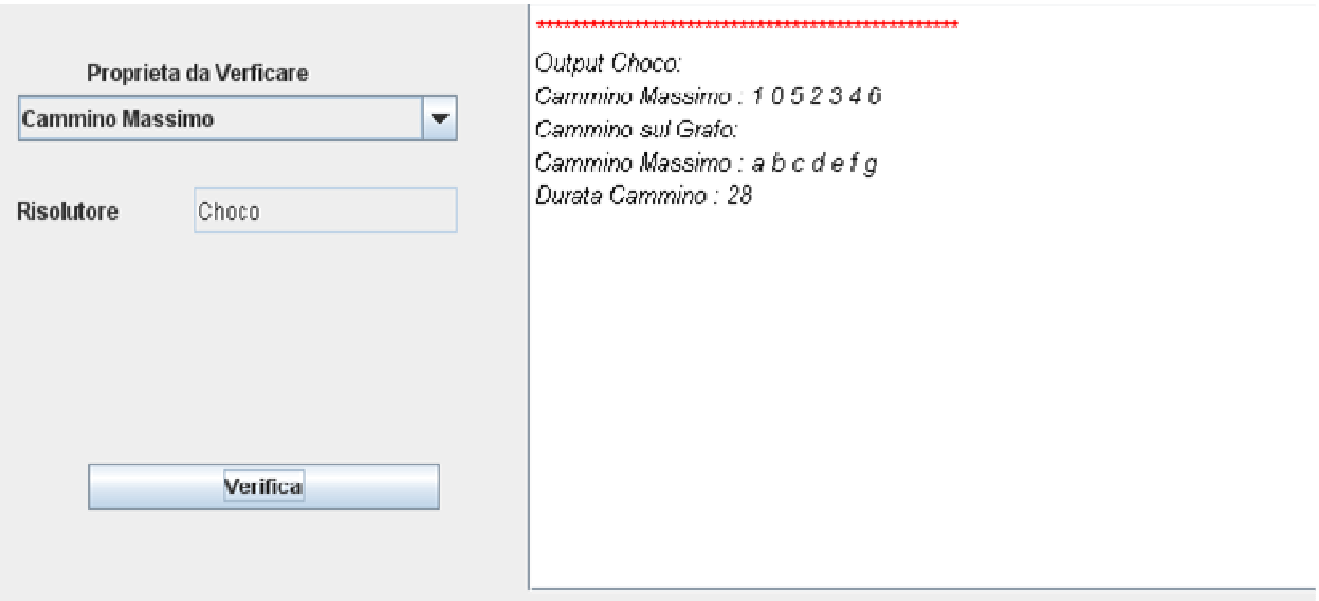
Esempio 1: grafo con un ciclo interno





NOME ATTIVITA	INIZIO-FINE	DURATA	COSTO	SUCCESSORI	DESCRIZIONE
a	Inizio	1	1	b	
b		2	2	c	
c		3	3	d	
d		4	4	e	
e		5	5	b f	
f		6	6	g	
g	Fine	7	7		

L'output di Choco:



Matrice *grafo*

0	2
1	2
1	5
2	3
3	4
4	1
5	6

ListaAttività

a	e	b	c	d	f	g

Vettore *v*

1	0	5	2	3	4	6

La cella i-sima del vettore *v* si riferisce alla riga i-sima della matrice *grafo*
Il valore j di ogni cella della matrice grafo indica la j-ima cella del vettore *ListaAttività*

L’output di NuSMV:

a : Nessun ciclo - Attivita Iniziale

e : null-- specification ((diagra.stato = e & (diagra.evento = eb | diagra.evento = ef)) -> X (G diagra.stato != e)) is false

Trace Description: LTL Counterexample

-- Loop starts here

  diagra.stato = e

-> Input: 1.2 <-

  diagra.stato = b

-> Input: 1.3 <-

  diagra.stato = c

-> Input: 1.4 <-

  diagra.stato = d

-> Input: 1.5 <-

  diagra.stato = e

null

b : null-- specification ((diagra.stato = b & diagra.evento = bc) -> X (G diagra.stato != b)) is false

Trace Description: LTL Counterexample

-- Loop starts here

  diagra.stato = b

-> Input: 1.2 <-

  diagra.stato = c

-> Input: 1.3 <-

  diagra.stato = d

-> Input: 1.4 <-

  diagra.stato = e

-> Input: 1.5 <-

  diagra.stato = b

null

c : null-- specification ((diagra.stato = c & diagra.evento = cd) -> X (G diagra.stato != c)) is false

Trace Description: LTL Counterexample

-- Loop starts here

  diagra.stato = c

-> Input: 1.2 <-

  diagra.stato = d

-> Input: 1.3 <-

  diagra.stato = e

-> Input: 1.4 <-

  diagra.stato = b

-> Input: 1.5 <-

  diagra.stato = c

null

d : null-- specification ((diagra.stato = d & diagra.evento = de) -> X (G diagra.stato != d)) is false

Trace Description: LTL Counterexample

-- Loop starts here

  diagra.stato = d

-> Input: 1.2 <-

  diagra.stato = e

-> Input: 1.3 <-

  diagra.stato = b

-> Input: 1.4 <-

  diagra.stato = c

-> Input: 1.5 <-

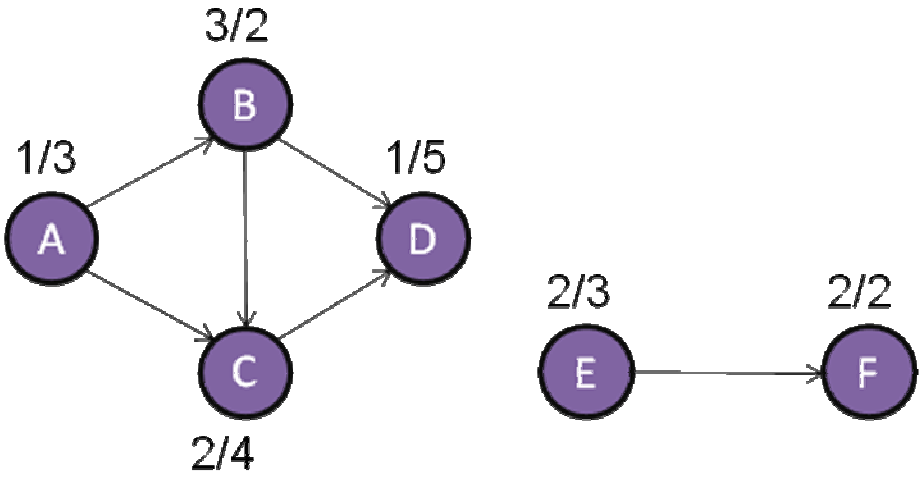
  diagra.stato = d

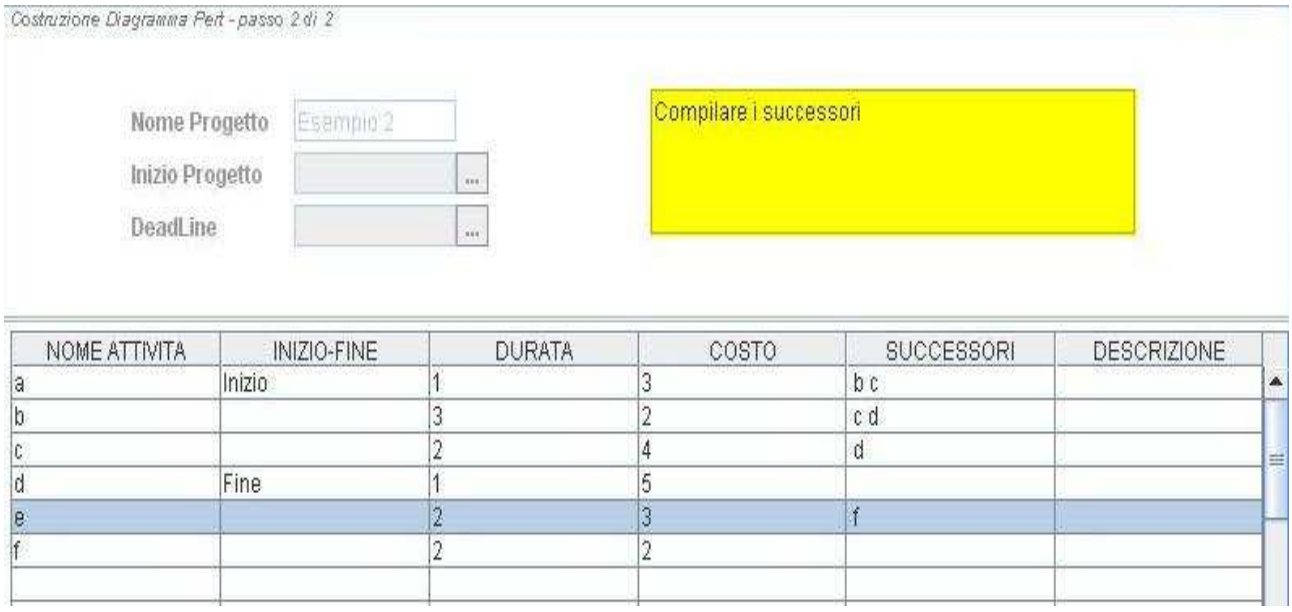
null

f : null-- specification ((diagra.stato = f & diagra.evento = fg) -> X (G diagra.stato != f)) is true

g : Nessun ciclo - Attivita Finale

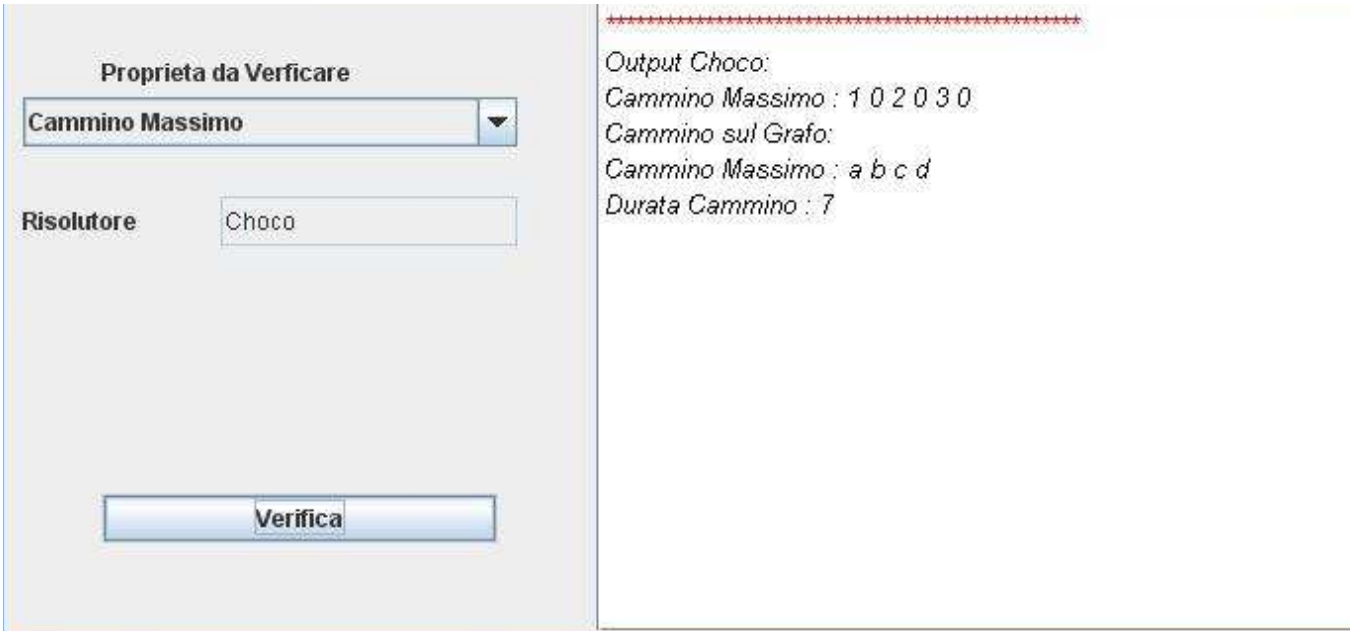
Esempio 2: grafo spezzato





NOME ATTIVITA	INIZIO-FINE	DURATA	COSTO	SUCCESSORI	DESCRIZIONE
a	Inizio	1	3	b c	
b		3	2	c d	
c		2	4	d	
d	Fine	1	5		
e		2	3	f	
f		2	2		

L'output di Choco:



Matrice *grafo*

0	1
0	2
1	2
3	4
1	5
2	5

ListaAttività

a	b	c	e	f	d

Vettore *v*

1	0	2	0	3	0

La cella i-sima del vettore *v* si riferisce alla riga i-sima della matrice *grafo*
Il valore j di ogni cella della matrice grafo indica la j-ima cella del vettore *ListaAttività*

L’output di NuSMV:

Verifica Aciclicita Grafo

a : Nessun ciclo - Attivita Iniziale

b : null-- specification ((diagra.stato = b & (diagra.evento = bc | diagra.evento = bd)) -> X (G diagra.stato != b)) is true

c : null-- specification ((diagra.stato = c & diagra.evento = cd) -> X (G diagra.stato != c)) is true

e : null-- specification ((diagra.stato = e & diagra.evento = ef) -> X (G diagra.stato != e)) is true

f : nullnull

d : Nessun ciclo - Attivita Finale