

# Progetto IA: Graphlog

Un tool in prolog/JS per l'analisi dei grafi

De Zuane Davide & El Mechri Rahmi



# Idea

- ▶ Utilizzare prolog per analizzare i grafi
- ▶ Rendere il progetto disponibile sul web
- ▶ Fornire un'interfaccia user friendly
- ▶ Unire le conoscenze di IA con Ricerca Operativa

# Natura dell'Ambiente

## Descrizione

- ▶ Ambiente: Teoria dei grafi
- ▶ Metriche di performance: tempi di esecuzione
- ▶ Attuatori: interprete prolog
- ▶ Sensori: interfaccia utente



## Proprietà

- ▶ Completamente Osservabile
- ▶ Agente singolo
- ▶ Deterministico
- ▶ Episodico
- ▶ Semi-statico
- ▶ Discreto

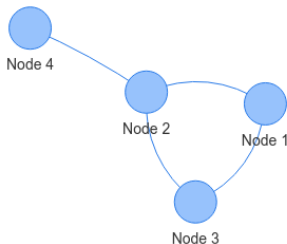
# Agente

- ▶ Simple reflex
- ▶ Percezione: grafo caricato dall'utente
- ▶ Azione: calcolo principali caratteristiche del grafo

Abbiamo realizzato una Single Page Application. Per realizzarla ci siamo serviti delle seguenti librerie:

- ▶  Tau-prolog: libreria javascript interprete prolog
- ▶  Vis.js: libreria javascript per la visualizzazione di grafi

```
function addNode(node) {  
  data.nodes.add(node);  
  //Add to Prolog KB  
  pl_kb_nodes_string += "node("+node.id+")."  
  session.consult(pl_kb_nodes_string+pl_kb_edges_string, {  
    success: function () {  
      console.log("Node added to KB successfully");  
      session.query("node(X).", {  
        success: function (goal) {  
          console.log("Query parsing went well");  
          session.answers(puri);  
        },  
        error: function (err) {  
          console.log("Query parsing went bad");  
        },  
      });  
    },  
    error: function (err) {  
      console.log("Node not added");  
    },  
  });  
}
```



# Knowledge Base

La Knowledge Base è definita da due parti.

## Fatti

I fatti vengono generati dinamicamente a partire dal grafo definito dall'utente, e sono definiti nel seguente modo:

- ▶ `node(x) .`
- ▶ `edge(x,y) .`

## Regole

Le regole vengono importate da un file prolog presente nel web server. Nel definire le regole abbiamo considerato solamente grafi simmetrici, tramite le regole:

- ▶ `edge_s(X,Y) :- edge(Y,X) .`
- ▶ `connected(X,Y) :- edge_s(X,Y) ; edge(X,Y) .`

# Regole - 1

## Stable set

```
stable_set([]).
stable_set([H]) :- node(H).
stable_set([H|T]) :- list_node(X), subset(X, [H|T]), disconnected(H, T), stable_set(T).
stable_set([H|T]) :- list_node([H|T]), disconnected(H, T), !.
maximum_stable_set(X) :- setof(Z, stable_set(Z), S), maximum_list_in_lists(S, X), !.
stable_set_of_cardinality(X, C) :- stable_set(X), list_length(X, Z), Z==C, !.
```

## Matching

```
matching(E) :- setof(X, edge_array(X), S), subset(S, E), arrays_dont_intersect(E).
edges_subset(E) :- setof(X, edge_array(X), S), subset(S, E).
arrays_dont_intersect([H]).
arrays_dont_intersect([H|[H1|T]]) :- intersection(H, H1, X), list_length(X, N), N==0, arrays_dont_intersect([H|T]), arrays_dont_intersect([H1|T]).
maximum_matching(M) :- setof(X, matching(X), S), maximum_list_in_lists(S, M).
```

# Regole - 2

## Edge Cover

```
edge_cover(X) :- setof(Y, edge_array(Y), E), subset(E,X), covered_nodes(X, N), list_node(L), same(L,N).
covered_nodes(E, N) :- elements_union(E, X), sort(X, N).
elements_union([H|T], X) :- elements_union_steps(T, H, X).
elements_union_steps([H], U, X) :- append(H, U, X).
elements_union_steps([H|T], U, X) :- append(H, U, Z), elements_union_steps(T, Z, X).
minimum_edge_cover(E) :- setof(X, edge_cover(X), S), minimum_list_in_lists(S, E).
```

## Vertex Cover

```
vertex_cover(X) :- list_node(L), subset(L, X), setof(Y, edge_array(Y), E), edge_covered_by_nodes(X,E).
edge_covered_by_nodes(X, [H]) :- subtract(H,X,S), list_lenght(S,N), N=<1.
edge_covered_by_nodes(X, [H|T]) :- subtract(H,X,S), list_lenght(S,N), N=<1, edge_covered_by_nodes(X, T).
minimum_vertex_cover(V) :- setof(X, vertex_cover(X), S), minimum_list_in_lists(S, V).
```



# Regole - 3

## Eulerian

```
even(X) :- Z is mod(X, 2), Z == 0.  
eulerian([H]) :- degree(H, N), even(N).  
eulerian([H|T]) :- connected_graph([H|T]), degree(H, N), even(N), eulerian(T), !.
```

## Hamiltonian

```
hamiltonian([H|T], P) :- check_hamiltonian_cycles(H, T, P).  
check_hamiltonian_cycles(X, [H|T], Y) :- path(X, H, P), n_nodes(N), list_length(P, N), last(Z, P), connected(Z, X), append(P, [X], Y), !.  
check_hamiltonian_cycles(X, [H|T], P) :- check_hamiltonian_cycles(X, T, P).
```

# Regole - 4

## Tree

```
tree([H|T]) :- n_nodes(N), n_edges(Z), Y is N-1, Z==Y, connected_graph([H|T]).
```

## Biparted

```
disconnected_graph([H]) :- node(H).  
disconnected_graph([H|T]) :- list_node(L), subset(L, [H|T]), disconnected(H, T), disconnected_graph(T).  
biparted(Z) :- list_node(L), disconnected_graph(X), subtract(L,X,Y), disconnected_graph(Y), Z=[X,Y], !.
```

## Regole - 5

Abbiamo inoltre definito le seguenti regole che fungono da **utility** per le altre:

- ▶ `intersection\3`
- ▶ `subtraction\3`
- ▶ `last\2`
- ▶ `same\2`
- ▶ `ordered\1`
- ▶ `arrays_dont_intersect\1`
- ▶ `subset\2`

# Javascript

Abbiamo dovuto realizzare le seguenti funzioni in Javascript:

- ▶ Parsare un file JSON contente i fatti.
- ▶ Importare le regole dal file prolog.
- ▶ Creare il grafo in vis.js a partire dai fatti.
- ▶ Interfacciare i componenti dell'interfaccia alle query in prolog.
- ▶ Iniettare i risultati all'interno dell'interfaccia
- ▶ Creare un grafo randomico.

# GUI - 1

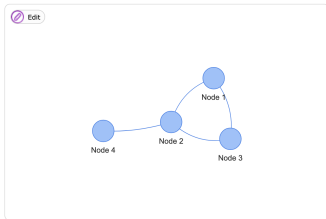
ALCHEMISTS



## GraphLog

### Graph Visualization

The visualization of the graph with Vis.js



### Action

Specify edges and nodes in JSON notation

Nodes

```
[
  {"id":1,"label":"Node 1"},
  {"id":2,"label":"Node 2"},
  {"id":3,"label":"Node 3"},
  {"id":4,"label":"Node 4"},
  {"id":5,"label":"Node 5"},
  {"id":6,"label":"Node 6"},
  {"id":7,"label":"Node 7"}
]
```

Edges

```
[
  {"from":1,"to":2},
  {"from":1,"to":3},
  {"from":1,"to":4},
  {"from":3,"to":4},
  {"from":4,"to":7},
  {"from":5,"to":7},
  {"from":2,"to":6}
]
```

Done

Random

# GUI - 2

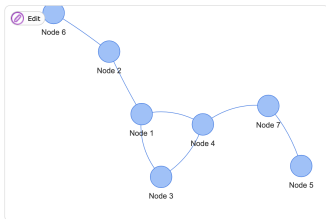
ALCHEMISTS



## GraphLog

### Graph Visualization

The visualization of the graph with Vis.js



Structure ? Query Other

Numero di nodi: 7

Numero di archi: 7

Nodo con stella massimo: 1 [2,3,4]

Nodo con stella minima: 5 [7]

Densità del Grafo: 0.33

Massimo Insieme Stabile: [1,5,6]

Massimo Abbinamento: [[1,2],[3,4],[5,7]]

Minima Copertuna con Archi: [[1,2],[2,6],[3,4],[5,7]]

# GUI - 3

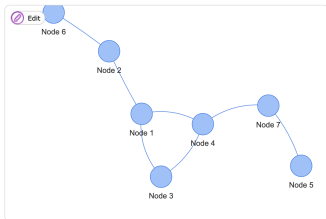
ALCHEMISTS



## GraphLog

### Graph Visualization

The visualization of the graph with Vis.js



Structure Query Other

Cammino minimo tra due nodi

$P = [1, 4, 7]$

Path

Ammette ciclo Euleriano?

false

Check

Ammette ciclo Hamiltoniano?

false

Check

Il grafo è un Albero?

false

Check

Bipartito?

false

Check

# Considerazioni

Prevediamo l'aggiunta delle seguenti funzionalità, già parzialmente implementate:

- ▶ Rendere l'ambiente semi-dinamico (aggiunta dinamica di nodi e archi)
- ▶ Analisi di grafi orientati
- ▶ Analisi di grafi pesati
- ▶ Visualizzazione dei risultati delle query sul grafo
- ▶ Animazione esecuzione query
- ▶ Ottimizzazione delle query



# Ringraziamenti

**Grazie per l'attenzione!**