Finding similar/dissimilar Solutions with ASP

Philipp Wanko

December 8, 2015

Content

Problem definition

Motivation

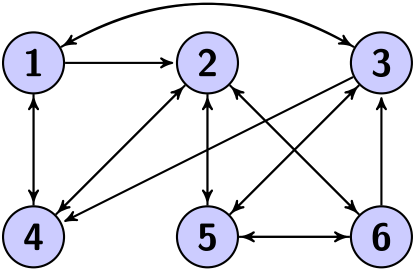
- subset of good diverse/similar solutions for decision-making
- Design space exploration
- Product configuration
- Planning
- Phylogeny reconstruction

Example: Hamiltonian cycle

```
% Generate
1{cycle(X,Y) : edge(X,Y)}1
:- node(X).
1{cycle(X,Y) : edge(X,Y)}1
:- node(Y).

% Define
reached(Y) :- cycle(1,Y).
reached(Y) :- cycle(X,Y);
reached(X).

% Test
:- node(Y), not reached(Y).
```



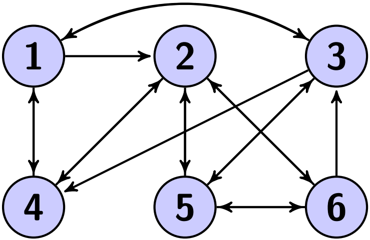
Example: Hamiltonian cycle

```
% Generate
1{cycle(X,Y) : edge(X,Y)}1 :- node(X).
1{cycle(X,Y) : edge(X,Y)}1 :- node(Y).

% Define
reached(Y) :- cycle(1,Y).
reached(Y) :- cycle(X,Y); reached(X).

% Test

:- node(Y), not reached(Y).
```



0 cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)

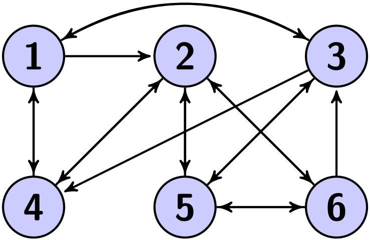
Example: Hamiltonian cycle

```
% Generate
1{cycle(X,Y) : edge(X,Y)}1 :- node(X).
1{cycle(X,Y) : edge(X,Y)}1 :- node(Y).


% Define
reached(Y) :- cycle(1,Y).
reached(Y) :- cycle(X,Y); reached(X).


% Test

:- node(Y), not reached(Y).
```



0. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
1. cycle(1,3) cycle(3,5) cycle(5,6) cycle(6,2) cycle(2,4) cycle(4,1)

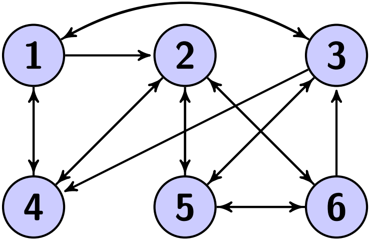
Example: Hamiltonian cycle

```
% Generate
1{cycle(X,Y) : edge(X,Y)}1 :- node(X).
1{cycle(X,Y) : edge(X,Y)}1 :- node(Y).


% Define
reached(Y) :- cycle(1,Y).
reached(Y) :- cycle(X,Y); reached(X).


% Test

:- node(Y), not reached(Y).
```



0. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
1. cycle(1,3) cycle(3,5) cycle(5,6) cycle(6,2) cycle(2,4) cycle(4,1)
2. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)

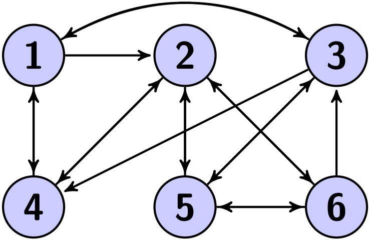
Example: Hamiltonian cycle

```
% Generate
1{cycle(X,Y) : edge(X,Y)}1 :- node(X).
1{cycle(X,Y) : edge(X,Y)}1 :- node(Y).


% Define
reached(Y) :- cycle(1,Y).
reached(Y) :- cycle(X,Y); reached(X).


% Test

:- node(Y), not reached(Y).
```



0. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
1. cycle(1,3) cycle(3,5) cycle(5,6) cycle(6,2) cycle(2,4) cycle(4,1)
2. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)
3. cycle(1,2) cycle(2,6) cycle(6,3) cycle(3,5) cycle(5,4) cycle(4,1)

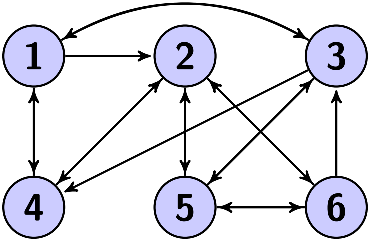
Example: Hamiltonian cycle

```
% Generate
1{cycle(X,Y) : edge(X,Y)}1 :- node(X).
1{cycle(X,Y) : edge(X,Y)}1 :- node(Y).


% Define
reached(Y) :- cycle(1,Y).
reached(Y) :- cycle(X,Y); reached(X).


% Test

:- node(Y), not reached(Y).
```



0. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
1. cycle(1,3) cycle(3,5) cycle(5,6) cycle(6,2) cycle(2,4) cycle(4,1)
2. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)
3. cycle(1,2) cycle(2,6) cycle(6,3) cycle(3,5) cycle(5,4) cycle(4,1)
4. cycle(1,2) cycle(2,6) cycle(6,5) cycle(5,3) cycle(3,4) cycle(4,1)

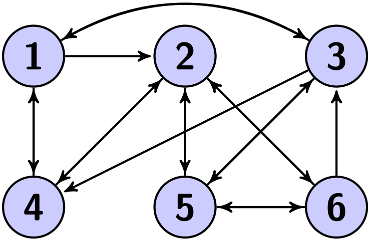
Example: Hamiltonian cycle

```
% Generate
1{cycle(X,Y) : edge(X,Y)}1 :- node(X).
1{cycle(X,Y) : edge(X,Y)}1 :- node(Y).


% Define
reached(Y) :- cycle(1,Y).
reached(Y) :- cycle(X,Y); reached(X).


% Test

:- node(Y), not reached(Y).
```



0. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
1. cycle(1,3) cycle(3,5) cycle(5,6) cycle(6,2) cycle(2,4) cycle(4,1)
2. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)
3. cycle(1,2) cycle(2,6) cycle(6,3) cycle(3,5) cycle(5,4) cycle(4,1)
4. cycle(1,2) cycle(2,6) cycle(6,5) cycle(5,3) cycle(3,4) cycle(4,1)
5. cycle(1,4) cycle(4,2) cycle(2,6) cycle(6,5) cycle(5,3) cycle(3,1)

Example: Distance

Distance function d in my example is percentage of different atoms.

Example: Distance

Distance function d in my example is percentage of different atoms.

2. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)
3. cycle(1,2) cycle(2,6) cycle(6,3) cycle(3,5) cycle(5,4) cycle(4,1)

Example: Distance

Distance function *d* in my example is percentage of different atoms.

2. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)
3. cycle(1,2) cycle(2,6) cycle(6,3) cycle(3,5) cycle(5,4) cycle(4,1)

Example: Distance

Distance function d in my example is percentage of different atoms.

2. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)
3. cycle(1,2) cycle(2,6) cycle(6,3) cycle(3,5) cycle(5,4) cycle(4,1)

$\hookrightarrow$ atoms of 2 solutions are 50% different, $d(2,3) = 50$.

Example: Set distance

Set distance Δ is maximum of pairwise distance d.
Given following set of solutions S:

Example: Set distance

Set distance Δ is maximum of pairwise distance d.
Given following set of solutions S:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)

Example: Set distance

Set distance Δ is maximum of pairwise distance d.
Given following set of solutions S:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
2. cycle(1,4) cycle(4,2) cycle(2,6) cycle(6,5) cycle(5,3) cycle(3,1)

Example: Set distance

Set distance Δ is maximum of pairwise distance d.
Given following set of solutions S:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
2. cycle(1,4) cycle(4,2) cycle(2,6) cycle(6,5) cycle(5,3) cycle(3,1)
3. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)

Example: Set distance

Set distance Δ is maximum of pairwise distance d.
Given following set of solutions S:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
2. cycle(1,4) cycle(4,2) cycle(2,6) cycle(6,5) cycle(5,3) cycle(3,1)
3. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)

Distances:

$d(1,2)$ 50%

Example: Set distance

Set distance Δ is maximum of pairwise distance d.
Given following set of solutions S:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
2. cycle(1,4) cycle(4,2) cycle(2,6) cycle(6,5) cycle(5,3) cycle(3,1)
3. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)

Distances:

$d(1,2)$ 50%
$d(1,3)$ 50%

Example: Set distance

Set distance Δ is maximum of pairwise distance d.
Given following set of solutions S:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
2. cycle(1,4) cycle(4,2) cycle(2,6) cycle(6,5) cycle(5,3) cycle(3,1)
3. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)

Distances:

$d(1,2)$ 50%
$d(1,3)$ 50%
$d(2,3)$ 100%

Example: Set distance

Set distance Δ is maximum of pairwise distance d.
Given following set of solutions S:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
2. cycle(1,4) cycle(4,2) cycle(2,6) cycle(6,5) cycle(5,3) cycle(3,1)
3. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)

Distances:

$d(1,2)$ 50%
$d(1,3)$ 50%
$d(2,3)$ 100%

$\hookrightarrow \Delta(S) = 100$

Problem Definition

Given ASP program P and set distance measure $\Delta : 2^{Sol(P)} \mapsto \mathbb{N}$:

Problem Definition

Given ASP program P and set distance measure $\Delta : 2^{Sol(P)} \mapsto \mathbb{N}$:

n k-similar/dissimilar solutions

Find a set S of n solutions of P where $\Delta(S) \leq k$ (resp. $\Delta(S) \geq k$)

Problem Definition

Given ASP program P and set distance measure $\Delta : 2^{Sol(P)} \mapsto \mathbb{N}$:

n k-similar/dissimilar solutions

Find a set S of n solutions of P where $\Delta(S) \leq k$ (resp. $\Delta(S) \geq k$)

n most similar/most dissimilar solutions

Find a set S of n solutions of P where $\Delta(S)$ is minimal (resp. maximal $\Delta(S)$)

Problem Definition

Given ASP program P and set distance measure $\Delta : 2^{Sol(P)} \mapsto \mathbb{N}$:

n k-similar/dissimilar solutions

Find a set S of n solutions of P where $\Delta(S) \leq k$ (resp. $\Delta(S) \geq k$)

n most similar/most dissimilar solutions

Find a set S of n solutions of P where $\Delta(S)$ is minimal (resp. maximal $\Delta(S)$)

Other similarity problems: k-similar/dissimilar solution, maximal n k-similar/dissimilar solutions, most similar/dissimilar solutions, k-similar/dissimilar set

Complexity

Problem	Complexity
n k-similar/dissimilar solutions	NP-complete
k-similar/dissimilar solution	NP-complete
maximal n k-similar/dissimilar solutions	$FNP//log$-complete
n most similar/dissimilar solutions	FP^{NP}-complete
similar/dissimilar solution	FP^{NP}-complete
k-similar/dissimilar set	NP-complete
k-similar/dissimilar optimal solutions	Σ_2^P-complete

Complexity

Problem	Complexity
n k-**similar/dissimilar solutions**	NP-complete
k-similar/dissimilar solution	NP-complete
maximal n k-similar/dissimilar solutions	$FNP//log$-complete
n **most similar/dissimilar solutions**	FP^{NP}-complete
similar/dissimilar solution	FP^{NP}-complete
k-similar/dissimilar set	NP-complete
k-**similar/dissimilar optimal solutions**	Σ_2^P-complete

Complexity

Problem	Complexity
n k-**similar/dissimilar solutions**	NP-complete
k-similar/dissimilar solution	NP-complete
maximal n k-similar/dissimilar solutions	$FNP//log$-complete
n **most similar/dissimilar solutions**	FP^{NP}-complete
similar/dissimilar solution	FP^{NP}-complete
k-similar/dissimilar set	NP-complete
k-**similar/dissimilar optimal solutions**	Σ_2^P-complete

- $\hookrightarrow$ challenging problems; need to find heuristics and approximations to handle complexity or accept restrictions.

Complexity

Problem	Complexity
n k-**similar/dissimilar solutions**	NP-complete
k-similar/dissimilar solution	NP-complete
maximal n k-similar/dissimilar solutions	$FNP//log$-complete
n **most similar/dissimilar solutions**	FP^{NP}-complete
similar/dissimilar solution	FP^{NP}-complete
k-similar/dissimilar set	NP-complete
k-**similar/dissimilar optimal solutions**	Σ_2^P-complete

- $\hookrightarrow$ challenging problems; need to find heuristics and approximations to handle complexity or accept restrictions.
- In practice mostly evolutionary/genetic problem specific algorithms for multiobjective optimization.

Main inspiration

- Ying Zhu and Miroslaw Truszczynski: *On Optimal Solutions of Answer Set Optimization Problems* (2013)

Main inspiration

- Ying Zhu and Miroslaw Truszczynski: *On Optimal Solutions of Answer Set Optimization Problems* (2013)
- Thomas Eiter, Esra Erdem, Halit Erdogan and Micheal Fink: *Finding Similar/Diverse Solutions in Answer Set Programming* (2011)

Main inspiration

- Ying Zhu and Miroslaw Truszczynski: *On Optimal Solutions of Answer Set Optimization Problems* (2013)
- Thomas Eiter, Esra Erdem, Halit Erdogan and Micheal Fink: *Finding Similar/Diverse Solutions in Answer Set Programming* (2011)

Three basic approaches are found in literature for ASP:

Main inspiration

- Ying Zhu and Miroslaw Truszczynski: *On Optimal Solutions of Answer Set Optimization Problems* (2013)
- Thomas Eiter, Esra Erdem, Halit Erdogan and Micheal Fink: *Finding Similar/Diverse Solutions in Answer Set Programming* (2011)

Three basic approaches are found in literature for ASP:

1. Offline method
2. Iterative method
3. Modifying solver branching heuristic

Clique approach

Overview

- Model solutions as vertices of graph with distances as labels of edges
- search for cliques in graph
- complete, correct
- easy to implement, versatile
- not efficient

Current implementation

- ASP problems can be normal logic programs or optimization problems in *asprin*-format
- solves n k-similar/dissimilar solutions and n most similar/most dissimilar solutions
- full python script
- distance function in python

Algorithm

Data: Distance function d, Problem P, distance k, number solutions n

Result: Set C of n solutions of P with $\Delta(S) \leq k$

S =getSolutions(P);

$V \leftarrow$Set of $|S|$ vertices, each element unique solution of P;

$E = \{(v_1, v_2) | v_1, v_2 \in V, v_1 \neq v_2, d(v_1, v_2) \leq k\}$;

$C \leftarrow$ clique with n vertices in $\langle V, E \rangle$;

return C

Getting solutions

S =getSolutions(P);

- P either normal logic program in ASP or optimization problem in asprin-format
- S contains all answer sets of P
- answer sets consist of shown atoms as *gringo Fun*-objects

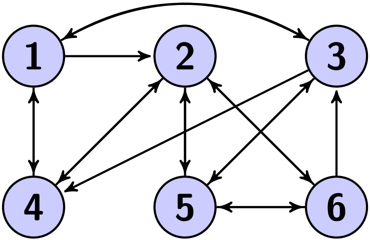
Getting solutions: Example

```
% Generate
1{cycle(X,Y) : edge(X,Y)}1 :- node(X).
1{cycle(X,Y) : edge(X,Y)}1 :- node(Y).


% Define
reached(Y) :- cycle(1,Y).
reached(Y) :- cycle(X,Y); reached(X).


% Test

:- node(Y), not reached(Y).
```



0. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
1. cycle(1,3) cycle(3,5) cycle(5,6) cycle(6,2) cycle(2,4) cycle(4,1)
2. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)
3. cycle(1,2) cycle(2,6) cycle(6,3) cycle(3,5) cycle(5,4) cycle(4,1)
4. cycle(1,2) cycle(2,6) cycle(6,5) cycle(5,3) cycle(3,4) cycle(4,1)
5. cycle(1,4) cycle(4,2) cycle(2,6) cycle(6,5) cycle(5,3) cycle(3,1)

Calculating cliques

$V \leftarrow$Set of $|S|$ vertices, each element unique solution of P;
$E = \{(v_1, v_2) | v_1, v_2 \in V, v_1 \neq v_2, d(v_1, v_2) \leq k\}$;
$C \leftarrow$ clique with n vertex in $\langle V, E\rangle$;

- first calculate pairwise distance between solutions
- build edges between all solutions with distances as labels
- add edges as instance to ASP clique program

Getting edges: Example

Distance function d in my example is percentage of different atoms.

Getting edges: Example

Distance function *d* in my example is percentage of different atoms.

2. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)
3. cycle(1,2) cycle(2,6) cycle(6,3) cycle(3,5) cycle(5,4) cycle(4,1)

Getting edges: Example

Distance function d in my example is percentage of different atoms.

2. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)
3. cycle(1,2) cycle(2,6) cycle(6,3) cycle(3,5) cycle(5,4) cycle(4,1)

Getting edges: Example

Distance function *d* in my example is percentage of different atoms.

2. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)
3. cycle(1,2) cycle(2,6) cycle(6,3) cycle(3,5) cycle(5,4) cycle(4,1)

↪ 3/6 of atoms are different; edge(2,3,50) is added to instance

Getting edges: Example

Distance function *d* in my example is percentage of different atoms.

2. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)
3. cycle(1,2) cycle(2,6) cycle(6,3) cycle(3,5) cycle(5,4) cycle(4,1)

$\hookrightarrow$ 3/6 of atoms are different; edge(2,3,50) is added to instance

Complete instance:

edge(0,1,83). edge(0,2,50). edge(0,3,83). edge(0,4,100).
edge(0,5,50). edge(1,2,66). edge(1,3,66). edge(1,4,83).
edge(1,5,100). edge(2,3,50). edge(2,4,50). edge(2,5,100).
edge(3,4,50). edge(3,5,83). edge(4,5,50).

Getting cliques: Example

```
#program clique_sim(n,k).

edge(X,Y,D):-edge(Y,X,D).
vert(X):-edge(X,_,_).
vert(Y):-edge(_,Y,_).

n{cl_vert(X):vert(X)}n.

cl_edge(X,Y):-cl_vert(X),cl_vert(Y),
                edge(X,Y,D),X<Y,D<=k.

:-cl_vert(X),cl_vert(Y),X<Y,
        0{cl_edge(X,Y):edge(X,Y,_)}0.
```

Getting cliques: Example

```
#program clique_sim(n,k).

edge(X,Y,D):-edge(Y,X,D).
vert(X):-edge(X,_,_).
vert(Y):-edge(_,Y,_).

n{cl_vert(X):vert(X)}n.

cl_edge(X,Y):-cl_vert(X),cl_vert(Y),
                edge(X,Y,D),X<Y,D<=k.

:-cl_vert(X),cl_vert(Y),X<Y,
        0{cl_edge(X,Y):edge(X,Y,_)}0.
```

For $k = 60$ and $n = 3$:

cl_vert(2), cl_vert(3), cl_vert(4)

Improvements

- optimal cliques
- only calculate subset of solutions
- iterate calculated solutions starting with number of required solutions
- add heuristic to enumerate more likely candidates

Getting optimal cliques: Example

```
#program clique_sim_opt(n).

...

cl_edge(X,Y,D):-cl_vert(X),cl_vert(Y),
                        edge(X,Y,D),X<Y.
...

#minimize { D@1,(cl_edge,X,Y): cl_edge(X,Y,D)}.
```

Getting optimal cliques: Example

```
#program clique_sim_opt(n).

...

cl_edge(X,Y,D):-cl_vert(X),cl_vert(Y),
                          edge(X,Y,D),X<Y.
...

#minimize { D@1,(cl_edge,X,Y): cl_edge(X,Y,D)}.
```

Optimal $k = 50$ for $n = 3$ with same solution:

cl_vert(2), cl_vert(3), cl_vert(4)

Iterative approach

Overview

- iteratively calculate solutions
- one call to the solver adds a solutions satisfying distance constraints
- not complete, correct
- easy to implement, only normal logic problems
- more efficient

Current implementation

- ASP problems can be normal logic programs
- solves n k-similar/dissimilar solutions and n most similar/most dissimilar solutions given a initial solution
- python script in logic program
- distance definition in ASP

Algorithm

Data:

- Solve.lp (calculates solution s of P)
- Distance.lp (calculates distances between set of solution S and s)
- Constraint.lp (eliminates solution s with distance $\Delta(S \cup \{s\}) > k$)
- number solutions n

Result: Set S of maximum n solutions of P with $\Delta(S) \leq k$

```
S = ∅;
for i = 1 to n do
    s ← Solve S Solve.lp Distance.lp Constraint.lp;
    if Unsat then
        break;
    end
    S = S ∪ s;
end
return S
```

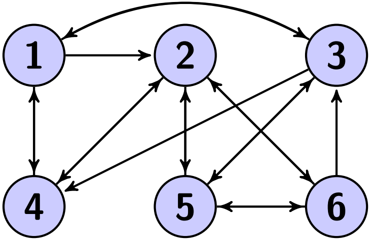
Solve.lp: Example

```
% Generate
1{cycle(X,Y) : edge(X,Y)}1 :- node(X).
1{cycle(X,Y) : edge(X,Y)}1 :- node(Y).

% Define
reached(Y) :- cycle(1,Y).
reached(Y) :- cycle(X,Y); reached(X).

% Test

:- node(Y), not reached(Y).
```



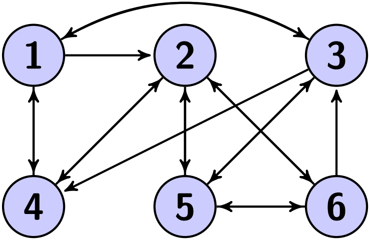
Solve.lp: Example

```
% Generate
1{cycle(X,Y) : edge(X,Y)}1 :- node(X).
1{cycle(X,Y) : edge(X,Y)}1 :- node(Y).


% Define
reached(Y) :- cycle(1,Y).
reached(Y) :- cycle(X,Y); reached(X).


% Test

:- node(Y), not reached(Y).
```



Additional definition of atoms that constitute a solution:

```
#program solve.
_solution(0,cycle(X,Y)):-cycle(X,Y).
#show cycle/2.
```

Each step a new solution 0 is calculated.

Distance.lp: Example

Following logic program saves solution and excludes it in the future ($S = S \cup s$):

Distance.lp: Example

Following logic program saves solution and excludes it in the future ($S = S \cup s$):

```
#program savesol(m).
_solution(m,X) :- X = @getSols(m).

#program deletemodel(m).
:- _solution(0,X) : X = @getSols(m);
   N #sum { 1,X: _solution(0,X) } N;
   N = @solSize(m).
```

Distance.lp: Example

Following logic program is grounded in each step for each element in S and calculates distance to s:

Distance.lp: Example

Following logic program is grounded in each step for each element in S and calculates distance to s:

```
#program  distance_prct(n,step).
_notsame12(step,n,0,X):-_step(step);_solution(n,X);
                        not _solution(0, X).
_notsame21(step,n,0,X):-_step(step);_solution(0,X);
                        not _solution(n, X).
_nratoms(step,n,0,N,K):-_step(step);N={_solution(n,X)};
                        K={_notsame12(step,n,0,A)}.
_nratoms(step,0,n,N,K):-_step(step);N={_solution(0,X)};
                        K={_notsame21(step,n,0,A)}.
_distance(step,n,0,K):- _step(step);_nratoms(step,n,0,N1,K1);
                        _nratoms(step,0,n,N2,K2);
                        K=@calcPrct(N1,K1,N2,K2).
```

Constraint.lp: Example

Following logic program is grounded in each step for each element in S to exclude s with $\Delta(S \cup s) > k$:

Constraint.lp: Example

Following logic program is grounded in each step for each element in S to exclude s with $\Delta(S \cup s) > k$:

```
#program constraint_sim(step,n,k).
:-_distance(step,n,0,X); X > k; _step(step).
```

Result: Example

All parts together with $k = 90$ and $n = 3$ yield the following results:

Result: Example

All parts together with $k = 90$ and $n = 3$ yield the following results:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)

Result: Example

All parts together with $k = 90$ and $n = 3$ yield the following results:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
2. cycle(1,3) cycle(3,5) cycle(5,6) cycle(6,2) cycle(2,4) cycle(4,1) _step(2) _distance(2,1,0,83)

Result: Example

All parts together with $k = 90$ and $n = 3$ yield the following results:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
2. cycle(1,3) cycle(3,5) cycle(5,6) cycle(6,2) cycle(2,4) cycle(4,1) _step(2) _distance(2,1,0,83)
3. cycle(1,2) cycle(2,6) cycle(6,3) cycle(3,5) cycle(5,4) cycle(4,1) _step(3) _distance(3,1,0,83) _distance(3,2,0,66)

Improvements

- use optimize statements to ensure least distance for next candidate
- no more need to specify k

Improvements

- use optimize statements to ensure least distance for next candidate
- no more need to specify k

Add following statement instead of Constraint.lp to the grounding and save the last model:

Improvements

- use optimize statements to ensure least distance for next candidate
- no more need to specify *k*

Add following statement instead of Constraint.lp to the grounding and save the last model:

```
#program opt_sim(step).
_maxdist(K,step):-K = #max{X:_distance(step,_,0,X)};
                  _step(step).
#minimize{K: _maxdist(K,step),_step(step)}.
```

Improvements: Example

Same example now without k and $n = 3$ yield the following results:

Improvements: Example

Same example now without k and $n = 3$ yield the following results:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)

Improvements: Example

Same example now without k and $n = 3$ yield the following results:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
2. cycle(1,4) cycle(4,2) cycle(2,6) cycle(6,5) cycle(5,3) cycle(3,1) _step(2) _distance(2,1,0,50)

Improvements: Example

Same example now without k and $n = 3$ yield the following results:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
2. cycle(1,4) cycle(4,2) cycle(2,6) cycle(6,5) cycle(5,3) cycle(3,1) _step(2) _distance(2,1,0,50)
3. cycle(1,2) cycle(2,6) cycle(6,3) cycle(3,5) cycle(5,4) cycle(4,1) _step(3) _distance(3,1,0,83) _distance(3,2,0,83)

Improvements: Example

Same example now without k and $n = 3$ yield the following results:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
2. cycle(1,4) cycle(4,2) cycle(2,6) cycle(6,5) cycle(5,3) cycle(3,1) _step(2) _distance(2,1,0,50)
3. cycle(1,2) cycle(2,6) cycle(6,3) cycle(3,5) cycle(5,4) cycle(4,1) _step(3) _distance(3,1,0,83) _distance(3,2,0,83)

Slight improvement in quality to $k = 83$ and better distance between 1 and 2 but not nearly optimal due to unfortunate start candidate.

asprin + Hclasp approach

Overview

- extend *asprin* preference framework with heuristic to enable similarity
- modify branching heuristic to find similar/dissimilar models from previous solutions
- no guarantees
- easy to implement, directly aids in finding solutions
- tampering with branching heuristics may decrease performance

Current implementation

- ASP problems can only be optimization problems in *asprin*-format
- approximates *n* most similar/most dissimilar solutions
- python script in logic program
- distance can only be expressed in _heuristic-atoms

Algorithm

- same branch and bound algorithm of *asprin*
- change branching heuristic with *hclasp* when optimal solution is found:

Algorithm

- same branch and bound algorithm of *asprin*
- change branching heuristic with *hclasp* when optimal solution is found:

Data: Set H of atoms of optmimal solution, step s
foreach $a \in H$ **do** Add atom _heuristic(_holds(a,0),true,s) ;

Algorithm

- same branch and bound algorithm of *asprin*
- change branching heuristic with *hclasp* when optimal solution is found:

Data: Set H of atoms of optmimal solution, step s
foreach $a \in H$ **do** Add atom _heuristic(_holds(a,0),true,s) ;

- variable with highest value s is decided first and declared true, if possible
- *CDCL*-algorithm tries to pick same atoms from past optimal solutions, regarding newer solutions the most

Adding heuristic

If optimal solution is found, following logic program is added:

Adding heuristic

If optimal solution is found, following logic program is added:

```
#program dosimilar(m).
_heuristic(_holds(X,0),true,m) :- X=@getHolds().

#show _holds/2.
#show _heuristic/3.
```

Adding heuristic: Example

```
% Generate
1{ cycle(X,Y) : edge(X,Y) }1
:- node(X).
1{ cycle(X,Y) : edge(X,Y) }1
:- node(Y).

% Define
reached(Y) :- cycle(1,Y).
reached(Y) :- cycle(X,Y);
reached(X).

% Test
:- node(Y), not reached(Y).
```

```
%optimize
#preference(c1,less(weight)){
  V::cycle(X,Y) : cost(1,X,Y,V)
}.
#preference(c2,less(weight)){
  V::cycle(X,Y) : cost(2,X,Y,V)
}.
#preference(c3,less(weight)){
  V::cycle(X,Y) : cost(3,X,Y,V)
}.

#preference(all,pareto){
  name(c1); name(c2); name(c3)
}.

#optimize(all).
```

Adding heuristic: Example

Same example with 3 random cost function at the edges. Pareto optimal answers are:

Adding heuristic: Example

Same example with 3 random cost function at the edges. Pareto optimal answers are:

0. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)

Adding heuristic: Example

Same example with 3 random cost function at the edges. Pareto optimal answers are:

0. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
1. cycle(1,3) cycle(3,5) cycle(5,6) cycle(6,2) cycle(2,4) cycle(4,1)

Adding heuristic: Example

Same example with 3 random cost function at the edges. Pareto optimal answers are:

0. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
1. cycle(1,3) cycle(3,5) cycle(5,6) cycle(6,2) cycle(2,4) cycle(4,1)
2. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)

Adding heuristic: Example

Same example with 3 random cost function at the edges. Pareto optimal answers are:

0. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
1. cycle(1,3) cycle(3,5) cycle(5,6) cycle(6,2) cycle(2,4) cycle(4,1)
2. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)
3. cycle(1,2) cycle(2,6) cycle(6,3) cycle(3,5) cycle(5,4) cycle(4,1)

Adding heuristic: Example

Same example with 3 random cost function at the edges. Pareto optimal answers are:

0. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
1. cycle(1,3) cycle(3,5) cycle(5,6) cycle(6,2) cycle(2,4) cycle(4,1)
2. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)
3. cycle(1,2) cycle(2,6) cycle(6,3) cycle(3,5) cycle(5,4) cycle(4,1)
4. cycle(1,4) cycle(4,2) cycle(2,6) cycle(6,5) cycle(5,3) cycle(3,1)

Adding heuristic: Example

cycle/2 is in preference declaration which leads to rule:

```
_holds(for(cycle(X,Y)),0):-cycle(X,Y).
```

Adding heuristic: Example

cycle/2 is in preference declaration which leads to rule:

```
_holds(for(cycle(X,Y)),0):-cycle(X,Y).
```

Optimal solution in step 2:

cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)

Adding heuristic: Example

cycle/2 is in preference declaration which leads to rule:

```
_holds(for(cycle(X,Y)),0):-cycle(X,Y).
```

Optimal solution in step 2:

cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)

Adds heuristic:

_heuristic(_holds(for(cycle(6,3)),0),true,2)
_heuristic(_holds(for(cycle(5,6)),0),true,2)
_heuristic(_holds(for(cycle(1,4)),0),true,2)
_heuristic(_holds(for(cycle(2,5)),0),true,2)
_heuristic(_holds(for(cycle(4,2)),0),true,2)
_heuristic(_holds(for(cycle(3,1)),0),true,2)

Adding heuristic: Example

First three answers without heuristic:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)

Adding heuristic: Example

First three answers without heuristic:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
2. cycle(1,4) cycle(4,2) cycle(2,6) cycle(6,5) cycle(5,3) cycle(3,1)

Adding heuristic: Example

First three answers without heuristic:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
2. cycle(1,4) cycle(4,2) cycle(2,6) cycle(6,5) cycle(5,3) cycle(3,1)
3. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)

Adding heuristic: Example

First three answers without heuristic:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
2. cycle(1,4) cycle(4,2) cycle(2,6) cycle(6,5) cycle(5,3) cycle(3,1)
3. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)

Distances:

1,2 50%

Adding heuristic: Example

First three answers without heuristic:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
2. cycle(1,4) cycle(4,2) cycle(2,6) cycle(6,5) cycle(5,3) cycle(3,1)
3. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)

Distances:

1,2 50%

1,3 50%

Adding heuristic: Example

First three answers without heuristic:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
2. cycle(1,4) cycle(4,2) cycle(2,6) cycle(6,5) cycle(5,3) cycle(3,1)
3. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)

Distances:

1,2 50%

1,3 50%

2,3 100%

Adding heuristic: Example

First three answers without heuristic:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
2. cycle(1,4) cycle(4,2) cycle(2,6) cycle(6,5) cycle(5,3) cycle(3,1)
3. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)

Distances:

1,2 50%

1,3 50%

2,3 100%

$\hookrightarrow k = 100$ and $n = 3$

Adding heuristic: Example

First three answers with heuristic:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)

Adding heuristic: Example

First three answers with heuristic:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
2. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)

Adding heuristic: Example

First three answers with heuristic:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
2. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)
3. cycle(1,3) cycle(3,5) cycle(5,6) cycle(6,2) cycle(2,4) cycle(4,1)

Adding heuristic: Example

First three answers with heuristic:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
2. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)
3. cycle(1,3) cycle(3,5) cycle(5,6) cycle(6,2) cycle(2,4) cycle(4,1)

Distances:

1,2 50%

Adding heuristic: Example

First three answers with heuristic:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
2. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)
3. cycle(1,3) cycle(3,5) cycle(5,6) cycle(6,2) cycle(2,4) cycle(4,1)

Distances:

1,2 50%

1,3 83%

Adding heuristic: Example

First three answers with heuristic:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
2. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)
3. cycle(1,3) cycle(3,5) cycle(5,6) cycle(6,2) cycle(2,4) cycle(4,1)

Distances:

1,2 50%

1,3 83%

2,3 66%

Adding heuristic: Example

First three answers with heuristic:

1. cycle(1,4) cycle(4,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,1)
2. cycle(1,2) cycle(2,5) cycle(5,6) cycle(6,3) cycle(3,4) cycle(4,1)
3. cycle(1,3) cycle(3,5) cycle(5,6) cycle(6,2) cycle(2,4) cycle(4,1)

Distances:

1,2 50%

1,3 83%

2,3 66%

$\hookrightarrow k = 83$ and $n = 3$

Improvements

- heuristic modifying atoms regarding all previous solution
- dynamic heuristic

Benchmarks

Overview

Clique:

Overview

Clique:

- Calculating all solutions:
  - finds globally optimal clique
  - nlp and optimization
  - inefficient

Overview

Clique:

- Calculating all solutions:
  - finds globally optimal clique
  - nlp and optimization
  - inefficient
- Calculating solutions iterative:
  - no optimal clique
  - nlp and optimization
  - more efficient

Overview

Clique:

- Calculating all solutions:
    - finds globally optimal clique
    - nlp and optimization
    - inefficient
- Calculating solutions iterative:
    - no optimal clique
    - nlp and optimization
    - more efficient

Iterative:

- no globally optimal solutions
- not guaranteed to find solution
- only nlp
- fast

Overview

asprin+hclasp:

- approximation of optimal solutions
- no hard cutoff
- only optimization
- fast

Setup

- all benchmarks were run on Zuse with 2 cores exclusively
- tried to find dissimilar solutions
- Optimization problems (6000 sek timeout, 20 Gb memout):
  - Design space exploration
  - Benchmark suite from *asprin*-paper with Pareto preference statements
- Normal problems (2000 sek timeout, 20 Gb memout):
  - Hamilton cycle suite
  - Benchmark suite from *asprin*-paper without preference statements

Results

		$n = 3$ $k = 60$ Clique	$n = 3$ $k = 60$ Clique(iter)	$n = 3$ $k = 60$ Iter
Class	#ins	time(s)	time(s)	time(s)
DSE	500	**2779.55(453)**	2832.50(455)	
asprin-paper-opt	133	2713.82(58)	**1298.26(26)**	
Hamilton	474	1986.96(470)	1322.72**(275)**	**1193.70**(280)
asprin-paper-nlp	133	1911.83 (127)	1576.63(92)	**880.17(52)**

Results

		$n=3$ opt Clique		$n=3$ opt Iter		$n=3$ opt heur	
Class	#ins	time(s)	dist	time(s)	dist	time(s)	dist
DSE	500	2777.67 (453)	986			**2723.18 (447)**	**1043**
asprin-paper-opt	133	2722.65 (58)	425			**361.03 (4)**	**4769**
Hamilton	474	1995.78 (473)	63	**1223.83 (289)**	**201**		
asprin-paper-nlp	133	1912.03 (127)	159	**1130.57 (73)**	**579**		

Conclusion

Conclusion

- iterative approach much better performance for normal logic programs
- with tweaks, clique approach is useful in small examples and for getting a baseline
- heuristic approach promising for multiobjective optimization problems

Improvements

- chose different starting solutions parallel for iterative approaches
- generate different subsets of solutions parallel for clique approach
- improve performance of getting a solution:
  - decrease iterations for *asprin* with *hclasp*
  - improve finding similar solutions with clique(iterative) and iterative approach with *hclasp*

Conclusion

Thank you! Questions?