ASP Modelling 1

Víctor Gutiérrez-Basulto

List of Tasks

1 Graph Colouring **1**

2 Travelling Salesperson **1**

3 Sorority **7**

Please complete past exercises

If you did not finish past labs, please work on that first.

1 Graph Colouring

Following the steps discussed during the lecture, write and execute program named `graphcolour.lp` for addressing the problem of graph colouring.
Run `clingo -n 0 <program.lp>` to display all the stable models of your program.

2 Travelling Salesperson

Read the following excerpt from the textbook:

3.3 ADVANCED PROBLEM ENCODING

Finally, we consider the well-known traveling salesperson problem. The task is to decide whether there is a round trip visiting each node in a graph exactly once (also known as a Hamiltonian cycle) such that accumulated edge costs do not exceed some budget. We tackle a slightly more general variant of the problem by not a priori fixing the budget. Rather, we want to compute a round trip with a minimum budget.

For this purpose, let us reconsider Graph G_8 but associate costs with edges. Figure 3.8 shows the augmented graph from Figure 3.1. Symmetric edges have the same costs here, but differing costs would be possible as well.

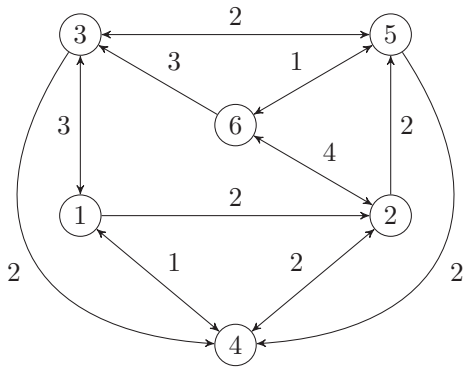

Figure 3.8: Graph G_8 from Figure 3.1 annotated with edge costs.

To accommodate edge costs, we augment the graph representation expressed in Program P_8 (cf. Listing 3.1) by the facts in Program P_{15} given in Listing 3.15.

Listing 3.15: Program P_{15} representing edge costs of Graph G_8 (`costs.lp`)

```
1  cost(1,2,2).  cost(1,3,3).  cost(1,4,1).
2  cost(2,4,2).  cost(2,5,2).  cost(2,6,4).
3  cost(3,1,3).  cost(3,4,2).  cost(3,5,2).
4  cost(4,1,1).  cost(4,2,2).
5  cost(5,3,2).  cost(5,4,2).  cost(5,6,1).
6  cost(6,2,4).  cost(6,3,3).  cost(6,5,1).
```

Program P_{15} contains an instance of `cost/3` for each (directed) edge in Figure 3.8.

As mentioned above, the first subproblem consists of finding a round trip, giving a candidate for a minimum-cost round trip. Following our *generate-and-test* methodology, we encode this subproblem via the following rules.

Listing 3.16: Program P_{16} addressing the round trip (Hamiltonian cycle) problem (`ham.lp`)

```
1  1 { cycle(X,Y) : edge(X,Y) } 1 :- node(X).
2  1 { cycle(X,Y) : edge(X,Y) } 1 :- node(Y).

4  reached(Y) :- cycle(1,Y).
5  reached(Y) :- cycle(X,Y), reached(X).
```

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

9  #hide.  #show cycle/2.
```

Solution candidates are represented by instances of predicate `cycle/2`, chosen among all edges of Graph G_8. Accordingly, Line 9 projects stable models on the respective instances.

The "generating" rules in Lines 1 and 2 make sure that each node in a graph must have exactly one outgoing and exactly one incoming edge, respectively. These edges are captured by means of predicate `cycle/2`. Let us make this more precise by inserting the available edges for node `1`. This yields the following instantiation of Lines 1 and 2:

```
1 { cycle(1,2), cycle(1,3), cycle(1,4) } 1.
1 { cycle(3,1), cycle(4,1) } 1.
```

The first rule groups all outgoing edges of node `1`, while the second does the same for its incoming edges. Together both rules provide us with six possibilities to get across node `1`.

The two rules in Lines 4 and 5 are "defining" rules, which (recursively) determine which nodes are reached by a cycle candidate produced in the "generating" part. Note that the rule in Line 4 builds on the assumption that the cycle "starts" at node `1`, that is, any successor `Y` of 1 is reached by the cycle. The second rule in Line 5 states that, from a reached node `X`, an adjacent node `Y` can be reached via a further edge in the cycle. Notably, the definition of `reached/1` in Lines 4 and 5 relies on an adequate treatment of positive recursion (among ground instances of `reached/1`).[2] That is, only derivable atoms are made true, while all others are set to false. This feature makes sure that all nodes are reached by a global cycle from node `1`, thus, excluding isolated subcycles. In fact, the "test" in Line 7 ensures that each node in a given graph is reached, that is, the instances of `cycle/2` in a stable model must be edges of a round trip.

Graph G_8 admits six round trips, as shown in Listing 3.17.

Listing 3.17: Grounding and solving Program $P_8 \cup P_{15} \cup P_{16}$

```
$ gringo graph.lp costs.lp ham.lp | clasp 0
clasp version 2.0.5
Reading from stdin
Solving...
Answer: 1
cycle(6,3) cycle(5,4) cycle(4,1) cycle(3,5) cycle(2,6) cycle(1,2)
Answer: 2
cycle(6,5) cycle(5,3) cycle(4,1) cycle(3,4) cycle(2,6) cycle(1,2)
Answer: 3
cycle(6,2) cycle(5,6) cycle(4,1) cycle(3,5) cycle(2,4) cycle(1,3)
Answer: 4
cycle(6,3) cycle(5,6) cycle(4,1) cycle(3,4) cycle(2,5) cycle(1,2)
Answer: 5
cycle(6,5) cycle(5,3) cycle(4,2) cycle(3,1) cycle(2,6) cycle(1,4)
Answer: 6
```

[2] Such positive recursion makes the resulting ground program non-tight (cf. Section 5.1).

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

Models       : 6
Time         : 0.001s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
```

We have so far ignored edge costs, and stable models of Program P_{16} correspond to round trips only. In order to find a minimum-cost journey among the six round trips of G_8, we add an "optimizing" part to Program P_{16}. This part is expressed by a single minimize statement given in Program P_{17}.

Listing 3.18: Program P_{17} minimizing edge costs for instances of `cycle/2` (`min.lp`)

```
11  #minimize [ cycle(X,Y) = C : cost(X,Y,C) ].
```

Here, edges belonging to the cycle are weighted according to their costs. After grounding, the minimization in Line 11 ranges over 17 instances of `cycle/2`, one for each (weighted) edge in G_8. For instance, instantiating the weighted (conditional) literal '`cycle(X,Y) = C : cost(X,Y,C)`' in view of the fact `cost(2,6,4)` in Listing 3.15 yields the ground weighted literal '`cycle(2,6) = 4`'.

Finally, we explain how the unique minimum-cost round trip (depicted in Figure 3.9) can be computed. The catch is that we are now interested in optimal stable models, rather than arbitrary ones. In order to determine the optimum, we can start by gradually decreasing the costs associated with stable models until we cannot find a strictly better one. In fact, *clasp* successively enumerates better stable models with respect to the provided optimization statements. Any stable model is printed as soon as it has been computed, and the last one is necessarily optimal. If there are multiple optimal stable models, an arbitrary one among them is computed. This proceeding is shown in Listing 3.19.

Listing 3.19: Grounding and solving Program $P_8 \cup P_{15} \cup P_{16} \cup P_{17}$

```
$ gringo graph.lp costs.lp ham.lp min.lp | clasp 0
clasp version 2.0.5
Reading from stdin
Solving...
Answer: 1
cycle(6,3) cycle(5,4) cycle(4,1) cycle(3,5) cycle(2,6) cycle(1,2)
Optimization: 14
Answer: 2
cycle(6,5) cycle(5,3) cycle(4,1) cycle(3,4) cycle(2,6) cycle(1,2)
Optimization: 12
Answer: 3
cycle(6,3) cycle(5,6) cycle(4,1) cycle(3,4) cycle(2,5) cycle(1,2)
Optimization: 11
OPTIMUM FOUND
```

```
Models       : 1
  Enumerated: 3
  Optimum    : yes
Optimization: 11
Time         : 0.001s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
```

Given that no answer is obtained after the third one, we know that 11 is the optimum value. However, there might be further stable models sharing the same optimum that have not yet been computed. In order to find them too, we can use the command line option `--opt-all=11` to enumerate all stable models having an objective value less or equal to 11. For the graph in Figure 3.8, the optimal stable model is unique. It is illustrated in Figure 3.9.

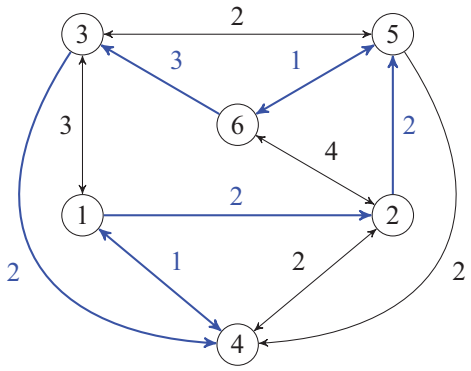

Figure 3.9: A minimum-cost round trip through Graph G_8.

Following the steps discussed during the lecture, write and execute program named `tsp.lp` for addressing the travelling salesperson.
Run `clingo -n 0 <program.pl>` to display **all** the stable models of your program.

3 Sorority

Exercise inspired by Chapter 1 of Introduction to Logic, by Michael Genesereth and Eric Kao (`http://ieeexplore.ieee.org/xpl/bkabstractplus.jsp?bkn=6812988`).
You know that:

- Dana likes Cody.
- Abby does not like Dana.
- Dana does not like Abby.
- Bess likes Cody or Dana.
- Abby likes everyone that Bess likes.
- Cody likes everyone who likes her.
- Nobody likes herself.

Describe the situation through an ASP program.

Run `clingo -n 0 <program.pl>` to display all the stable models of your program.
Confront those models with the four possible worlds presented in the following:

	Abby	Bess	Cody	Dana
Abby			✓	
Bess			✓	
Cody	✓	✓		✓
Dana			✓	

	Abby	Bess	Cody	Dana
Abby		✓	✓	
Bess			✓	
Cody	✓	✓		✓
Dana			✓	

	Abby	Bess	Cody	Dana
Abby			✓	
Bess			✓	
Cody	✓	✓		✓
Dana		✓	✓	

	Abby	Bess	Cody	Dana
Abby		✓	✓	
Bess			✓	
Cody	✓	✓		✓
Dana		✓	✓	