Modelling and Optimisation with Graphs

Özgür Akgün Fraser Dunlop Jessica Enright
Chris Jefferson **Ciaran McCreesh** Patrick Prosser
James Trimble

University of Glasgow
University of St Andrews
THE UNIVERSITY of EDINBURGH

Constraint Programming

- A declarative way of specifying (hard) problems.
- Variables, each with a domain of possible values.
  - Finite, usually integers or booleans.
  - Arrays and sets of integers also supported, either directly or indirectly.
  - Some solvers do floating point.
- Constraints.
  - No requirements of linearity, convexity, etc.
  - Rich global constraints, e.g. all different, regular, circuit.
- An objective.
  - Assign each variable a value from its domain, respecting all constraints.
  - Decide, enumerate, minimise a variable, or maximise a variable.
  - Can also produce Pareto fronts.

Solvers

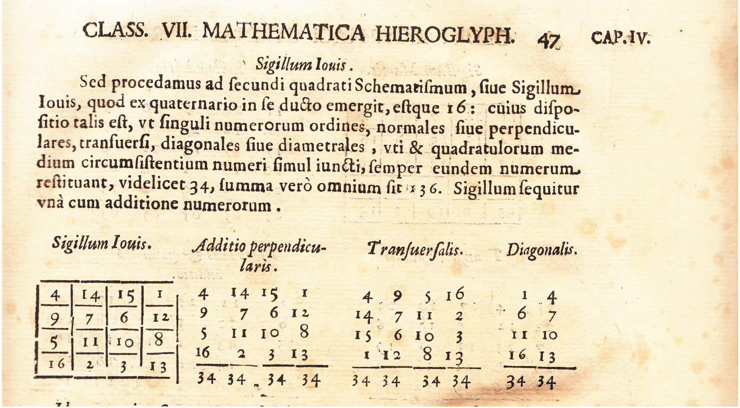
- Dedicated constraint programming solvers:
  - Gecode, Minion, Choco, OR-Tools, ...
  - Combine inference and intelligent backtracking search.
- Specialised solvers for more restricted settings:
  - SAT (boolean variables and constraints).
  - MIP (integer and real variables, linear inequalities).
  - Lots more...
- Or local search, which loses any guarantees of completeness.

High Level Modelling with Essence and Conjure

`http://conjure.readthedocs.io/en/latest/`

- Developed at St Andrews and York.
- Supports rich and nested structural types, such as sets of functions from sets of sets to partitions of matrices.
- Extensive automated reformulation.
- Targets CP, MIP, SAT, local search solvers.

Magic Squares



CLASS. VII. MATHEMATICA HIEROGLYPH. 47 CAP. IV.

Sigillum Iouis.

Sed procedamus ad ſecundi quadrati Schematiſmum, ſiue Sigillum Iouis, quod ex quaternario in ſe ducto emergit, eſtque 16: cuius diſpoſitio talis eſt, vt ſinguli numerorum ordines, normales ſiue perpendiculares, tranſuerſi, diagonales ſiue diametrales, vti & quadratulorum medium circumſiſtentium numeri ſimul iuncti, ſemper eundem numerum reſtituant, videlicet 34, ſumma verò omnium ſit 136. Sigillum ſequitur vnà cum additione numerorum.

Sigillum Iouis.

4	14	15	1
9	7	6	12
5	11	10	8
16	2	3	13

Additio perpendicularis.

4	14	15	1
9	7	6	12
5	11	10	8
16	2	3	13
34	34	34	34

Tranſuerſalis.

4	9	5	16
14	7	11	2
15	6	10	3
1	12	8	13
34	34	34	34

Diagonalis.

1	4
6	7
11	10
16	13
34	34

Page 47 of the second part of Vol. II of Athanasius Kircher's "Oedipus Aegyptiacus", published 1653. Scan by Feldkurat Katz. Public domain.

Magic Squares

```
given n : int(1..)

letting Index be domain int(1..n),
        Value be domain int(1..n**2)

find square : matrix indexed by [Index,Index] of Value,
     s : int(1..sum i : int(n**2+1-n..n**2) . i)

such that
    allDiff(flatten(square)),
    forAll r : Index . (sum c : Index . square[r,c]) = s,
    forAll c : Index . (sum r : Index . square[r,c]) = s,
    (sum d : Index . square[d,d]) = s,
    (sum d : Index . square[d,n+1-d]) = s
```

Magic Squares

```
$ echo "letting n be 4" > magic.param
$ ./conjure solve magic.essence magic.param
$ tail magic-magic.solution
 1  2 15 16
12 14  3  5
13  7 10  4
 8 11  6  9
```

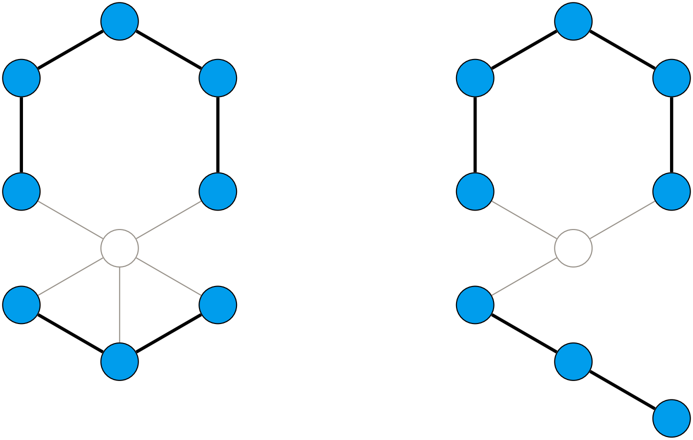
What About Graphs?

- How are they represented? Adjacency lists? Adjacency matrix?
- Some of the graphs we'd like to deal with are quite big.
- Expressing certain constraints (e.g. connectivity) by hand is difficult and heavily representation-dependent.
- It's very easy to end up with an $O(|V|^3)$ or $O(|V|^4)$ in the encoding size…

Modelling and Optimisation with Graphs

- Three year project led by Glasgow, together with St Andrews and Edinburgh.
- Working:
  - High level modelling for graphs, in Essence.
  - Better dedicated graph solvers.
- Ongoing:
  - Better compilation and reformulation.
  - Hybrid solving strategies.

Graphs by Example: Maximum Common Subgraphs



Graphs by Example: Maximum Common Subgraphs

- Finding the difference between two graphs comes down to finding as large a graph as possible that they both have in common. This is known as the *maximum common induced subgraph problem.*
- This concept generalises to n graphs.
- Application in metabolomics: we're given approximate molecular weights for the constituents of some compound, and we want to identify what the molecules are from a database. Sets of molecules with high similarities are much more likely to occur than unrelated molecules.

Graphs by Example: Maximum Common Subgraphs

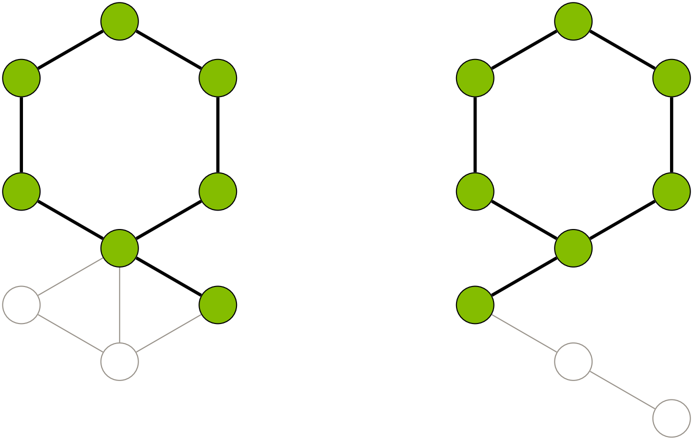
```
given n : int
given t : int
given G : matrix indexed by [int(1..t)] of
              graph of int(1..n)

find z : graph of int(1..n)
find F : matrix indexed by [int(1..t)] of
             function int(1..n) --> int(1..n)

such that forAll i : int(1..t) .
    subisomorphismInduced(z, G[i], F[i])

maximising |vertices(z)|
```

Graphs by Example: Maximum Common Subgraphs



Graphs by Example: Diseased Cows

- We have a graph of contacts (trade, or adjacent farms) between cattle in Scotland.
- If a disease outbreak occurs, we want to limit its spread.
- Can we vaccinate or screen on a small number of trade and contact routes?
- This comes down to deleting edges from a graph, to avoid having any large components.

Graphs by Example: Diseased Cows

```
given n : int
given k : int
given g : graph of int(1..n)

find deletions : set of (int(1..n), int(1..n))
find h         : graph of int(1..n)

such that h subgraph g
such that edges(h) = edges(g) - deletions
such that all[ |cc| < k | cc <- connectedComponents(h)]

minimising |deletions|
```

Graphs by Example: Kidney Exchange

- If two people both need kidney transplants, and have willing but incompatible donors, then they can exchange donors.
- Also, we can do this with cycles of three people.
- Also, we might have altruistic donors.
- Potential exchanges show up as 2-cycles, 3-cycles, etc in a graph.
- Given a set of pattern graphs, try to cover as much of the exchange graph as possible, not using any vertex more than once.

Graphs by Example: Kidney Exchange

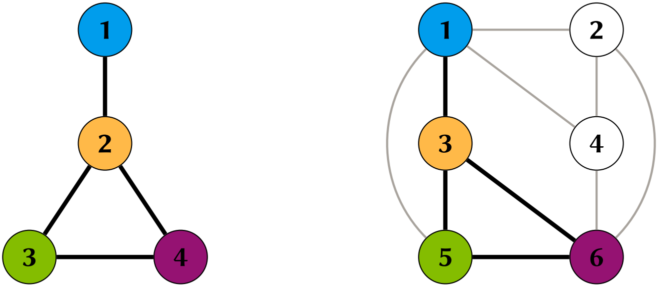
```
given np          : int
given max_pat_sz : int
given patterns   : matrix indexed by [int(1..np)] of
                       graph of int(1..max_pat_sz)
given benefit    : matrix indexed by [int(1..np)] of int
given tgt_sz     : int
given target     : graph of int(1..tgt_sz)
find map         : set of (int(1..np), function
                       int(1..max_pat_sz) --> int(1..tgt_sz))
such that forAll (i, f) in map .
    subisomorphism(patterns[i], target, f)
such that forAll {(i1,f1), (i2,f2)} subsetEq map .
    range(f1) intersect range(f2) = {}
maximising sum([benefit[i] | (i, f) <- map])
```

Unfortunately…

- Often orders of magnitude slower than dedicated solvers.
- Can only handle small graphs.

The Glasgow Subgraph Solver

https://github.com/ciaranm/glasgow-subgraph-solver



The Glasgow Subgraph Solver

https://github.com/ciaranm/glasgow-subgraph-solver

- The bestest subgraph isomorphism solver in the whole wide world.
- Support for non-induced subgraph isomorphism, induced subgraph isomorphism, graph homomorphisms, locally injective graph homomorphisms, clique.
- A bit like a constraint programming solver, but with specialised algorithms, data structures, and search strategies.

Unfortunately…

- It only supports subgraph finding, and basic labelling on vertices and edges.
- I *really* don't want to have to start implementing dozens of new constraints for it.
- Although it's theoretically possible to reduce non-subgraph constraints to clique, the encoding is huge and loses lots of helpful information.
    - (Exceptions apply, and this *can* be a really good idea occasionally.)
- For graph generation problems, we probably want to use another different solver.

Why Not Both?

- We could use a graph solver for graphy things, and a constraint programming solver for rich constraints.
- This can go in at least two ways:
  - Let a graph solver use a constraint programming solver to check a few side constraints. Useful for “find me a subgraph isomorphism that uses no more than three red vertices and at least two blue vertices”.
  - Use graph solvers to (dynamically?) generate parts of a constraint or MIP model. This is the state of the art for kidney exchange.
- The high level modelling suite of tools should take care of all of this for us.

The Easy Part

- The subgraph solver can output partial or full candidate assignments, or reduced domains.
- The constraint solver can treat these as additional constraints to a model.
  - Output one of "yes", or "no because ...".
  - Better: have an option for an "I can't easily tell, but I do know that ..." mode.
  - Fortunately, this is minimal effort in most constraint solvers.
- Three line shell script to glue the two together.

The Hard Part

- Which bits go where?

http://www.dcs.gla.ac.uk/~ciaran

ciaran.mccreesh@glasgow.ac.uk

University of Glasgow