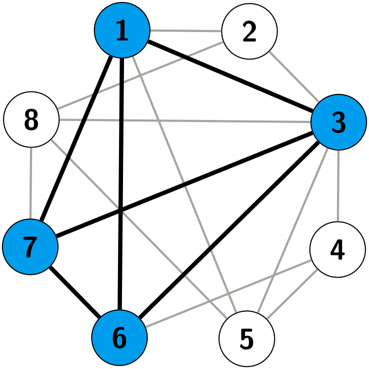
Solving Hard Graph Problems in Parallel

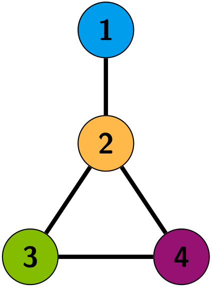
Ciaran McCreesh and Patrick Prosser

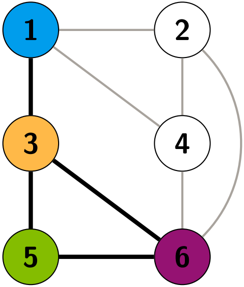
University of Glasgow

The Maximum Clique Problem

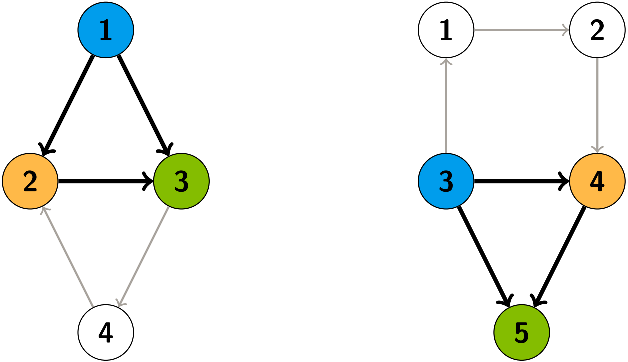


Subgraph Isomorphism

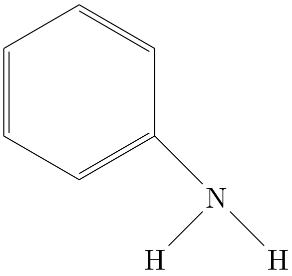


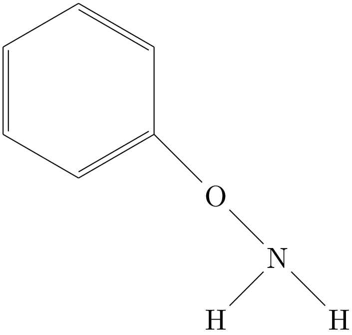


Maximum Common Subgraph



Maximum Common Connected Subgraph





Who Cares?

- Bioinformatics
- Chemistry
- Drug design
- Computer vision
- Pattern recognition
- Financial fraud detection
- Model checking
- Fault detection
- Law enforcement
- Kidney exchange
- Social network analysis
- Compilers
- Diseased cows
- Computer algebra
- Circuit design
- Network design

Practical Algorithms

- Real-world inputs rarely have nice properties (low treewidth, particular degree spreads that are polynomial, etc).
- Worst-case performance analysis tells us nothing.
- Constant factors matter.

Constraint Models

- We have some **variables**, each with a **domain**, and we want to give each variable a value from its domain.
  - Clique: a boolean variable for each vertex.
  - Subgraph isomorphism: a variable for each pattern vertex, with domains being target vertices.
- There are **constraints** between variables.
  - Clique: for each pair of non-adjacent vertices, at least one of the two variables must be false.
  - Subgraph isomorphism: all-different (injectivity), and adjacent pairs of vertices must be mapped to adjacent pairs of vertices.
- There is an **objective**.
  - Clique: set as many variables to true as possible.
  - Subgraph isomorphism: give each variable a value.

Preprocessing

- We want to **cross out values** from domains, until only one value is left in each.
- Subgraph isomorphism: high degree vertices cannot be mapped to low degree vertices.

Search

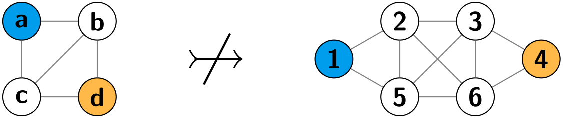
- Sometimes we have to **guess**: pick a variable x. Then for each value v_i in its domain in turn, see what happens if we force $x = v_i$.
- There are good heuristics telling us which variable to pick first.
- There are heuristics telling us which value to pick first, but this seems to be less reliable in general.

Inference

- After we guess an assignment, we can **infer additional deletions**. This can have a cascade effect.

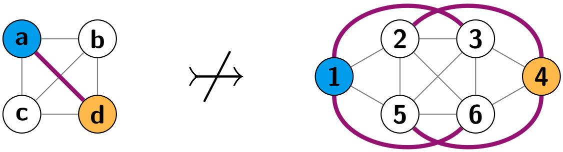
Implied Constraints for Subgraph Isomorphism

- Adjacent vertices must be mapped to adjacent vertices.
- Vertices that are distance 2 apart must be mapped to vertices that are within distance 2.
- Vertices that are distance k apart must be mapped to vertices that are within distance k.



Implied Constraints for Subgraph Isomorphism

- G^d is the graph with the same vertex set as G, and an edge between v and w if the distance between v and w in G is at most d.
- For any d, a subgraph isomorphism $i : P \rightarrowtail T$ is also a subgraph isomorphism $i^d : P^d \rightarrowtail T^d$.



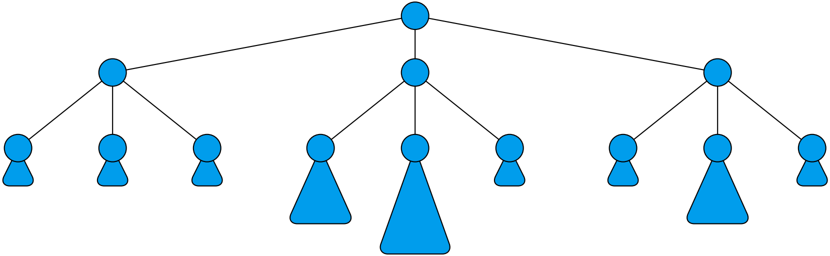
Implied Constraints for Subgraph Isomorphism

- We can do something stronger: rather than looking at distances, we can look at **(simple) paths**, and we can count how many there are.
- This is NP-hard in general, but only lengths 2 and 3 and counts of 2 and 3 are useful in practice.
- We construct these graph pairs once, at the top of search.

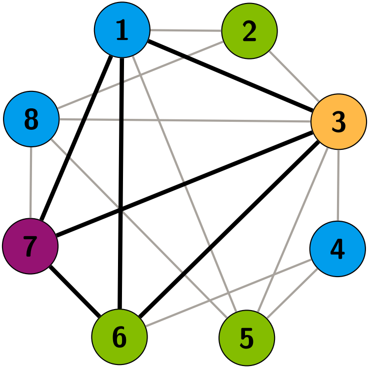
Backtracking Search as a Tree

- Sometimes we guess incorrectly, or there is no solution.
- When a variable's domain becomes empty, we fail, and **backtrack** one level and try something else.

Backtracking Search as a Tree

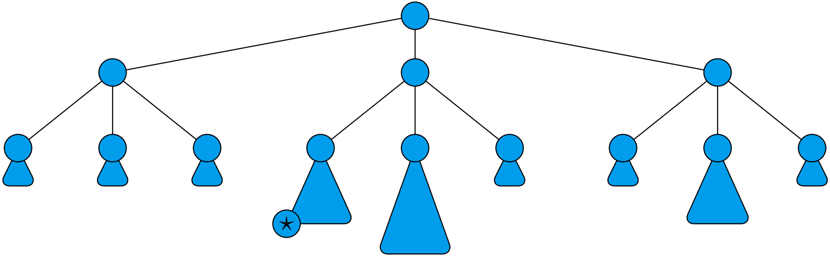


Branch and Bound

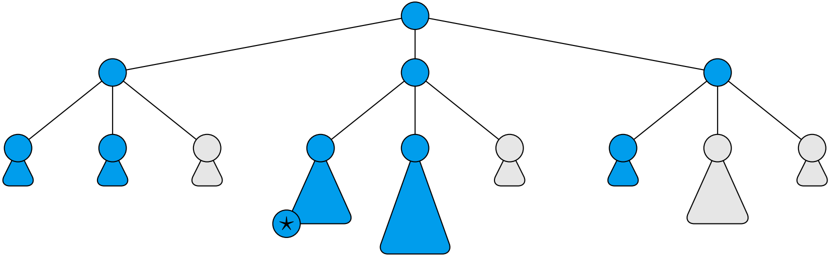


- For optimisation: keep track of the **best solution** we've found so far. If we can show we can't beat it, backtrack immediately.

Eliminable Sub-trees



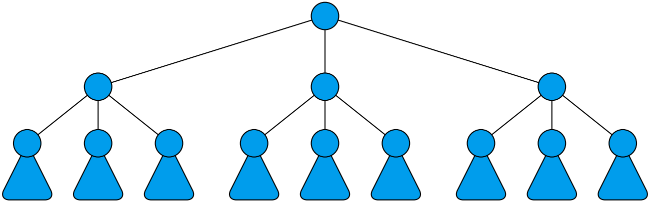
Eliminable Sub-trees



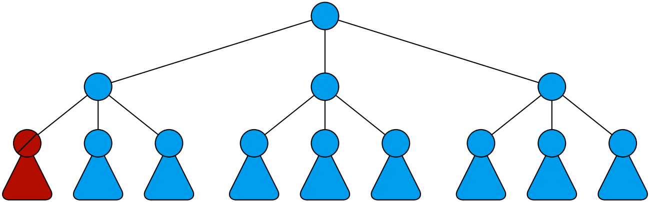
Backjumping

- When backtracking, see if the current assignment actually removed any values which could have helped prevent the failure. If not, **jump back** another step.

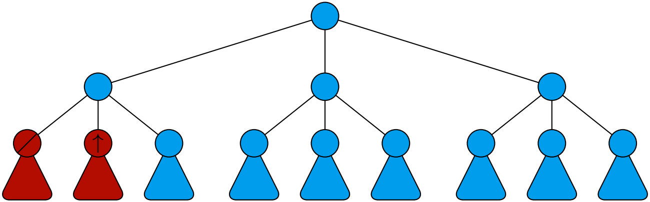
Backjumping as a Tree



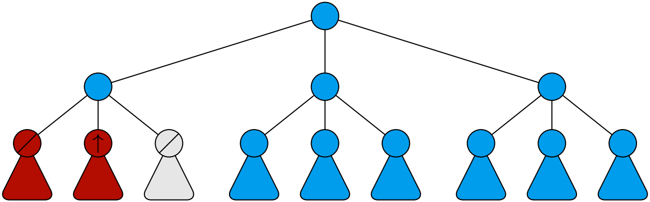
Backjumping as a Tree



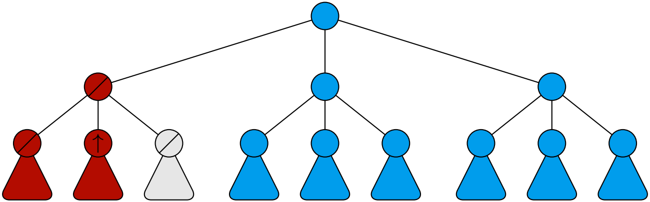
Backjumping as a Tree



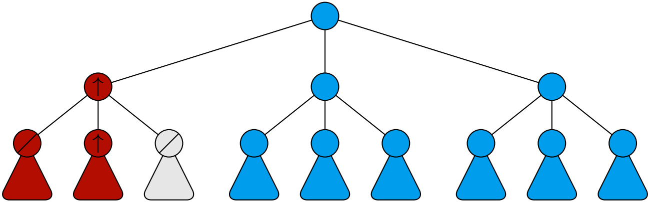
Backjumping as a Tree



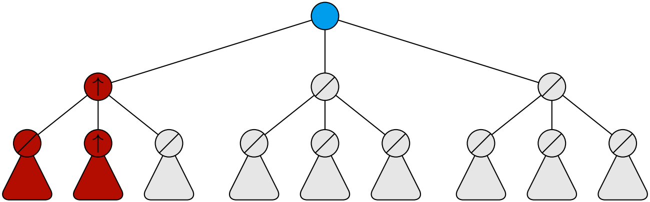
Backjumping as a Tree



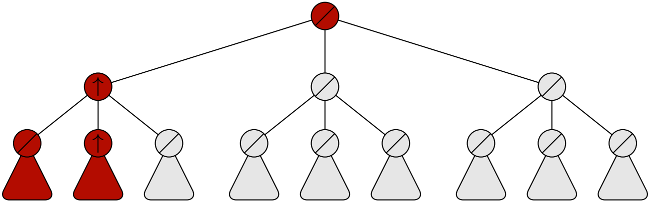
Backjumping as a Tree



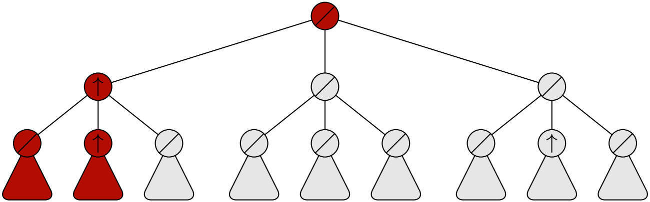
Backjumping as a Tree



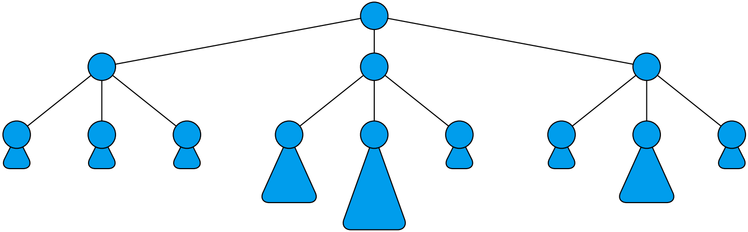
Backjumping as a Tree



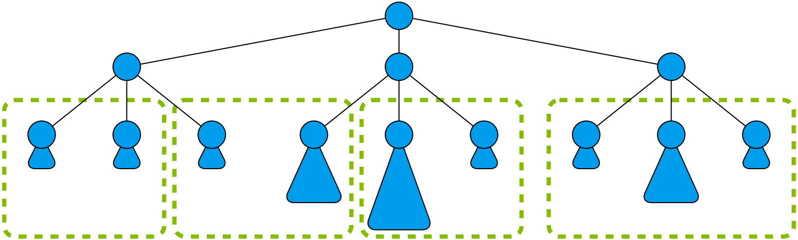
Backjumping as a Tree



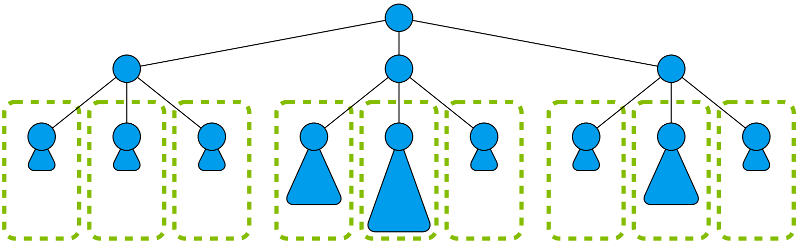
Thread-Parallel Tree Search



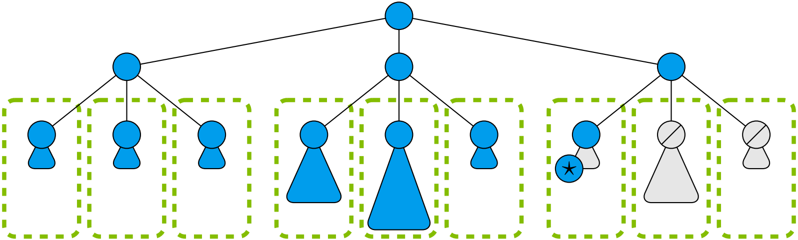
Thread-Parallel Tree Search



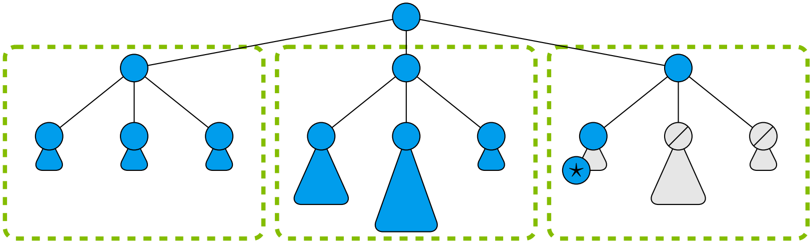
Thread-Parallel Tree Search



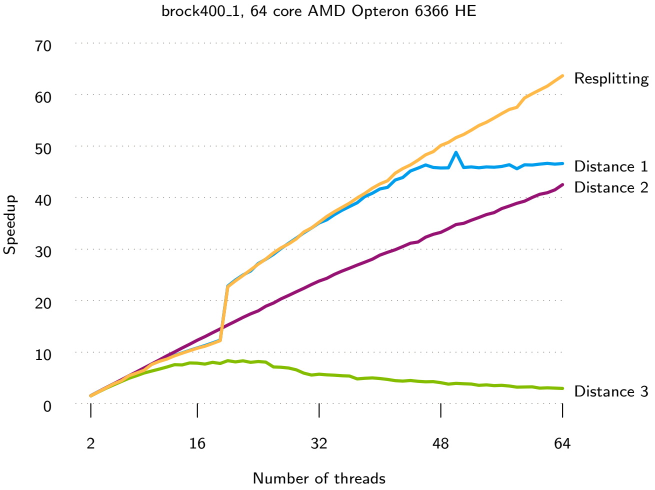
Parallel Search Order Matters



Parallel Search Order Matters



Parallel Search Order Matters



Parallel Search Order Matters

- Value-ordering heuristics tend to be **worst high up** the search tree.
- But depth-first searches commit completely to the first choice made. . .
- Discrepancy searches can avoid this problem by doing more work in total. Parallel search can give **similar benefits for free**.

Safety and Reproducibility

- My "wish list":
  1. Parallel search should not be substantially slower than sequential search.
  2. Adding more processors should not make things substantially worse.
  3. Running the same program twice on the same hardware should give similar runtimes.
- This is surprisingly tricky.
- On top of all that, we want to prioritise work stealing from where we're most likely to be wrong, or possibly from where we're most likely not to eliminate a subtree.

Parallel Backjumping as a Lazy Fold

- Lazily map each subproblem to Jump F **or** Fail F **or** Success.
- Lazily fold, starting with Fail $\{v\}$, as follows:

$$
\begin{aligned}
\_\_ \circledcirc \text{Success} &= \text{Success} \\
\_\_ \circledcirc \text{Jump } F &= \text{Jump } F \\
\text{Fail } F \circledcirc \text{Fail } G &= \text{Fail } (F \cup G)
\end{aligned}
$$

- If a Jump F occurs to the left of a Success, we have a bug.

Parallel Backjumping as a Lazy Fold

- When multiplying, if any item is 0, the result is 0.

$$\_\_ \times 0 = 0$$
$$0 \times \_\_ = 0$$

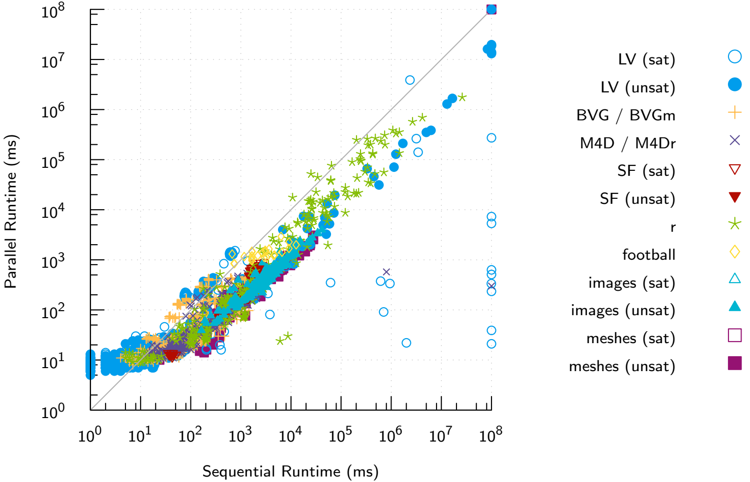
- Here, if any item is Success, the result is Success, and we do not need to evaluate the rest of the map.

$$\_\_ \circledcirc \text{Success} = \text{Success}$$

- If any item is Jump F, the result is either Jump F, or some Jump G or Success that is further to the left. We do not need to evaluate any item to the right.

$$\_\_ \circledcirc \text{Jump } F = \text{Jump } F$$

Parallel Search is Worth Doing



Describing and Implementing Parallel Search

- Implementing safe and reproducible parallel search by hand, even just for multi-core, is painful.
- Current high level approaches don't offer the properties we need.
- Is there a better way?

Symmetries

- Some graphs have known symmetries. Can we exploit this?
  - In some ways, maximum clique is just a completely symmetric version of maximum common subgraph.
- What about if we have to detect the symmetries ourselves dynamically?

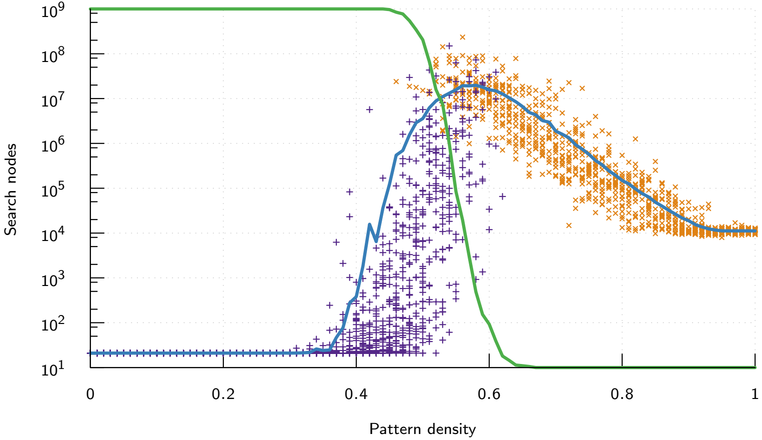
Explaining Failures

- Backjumping works because when we fail, we work out why, and use that to backtrack further.
- But then we throw that information away. . .

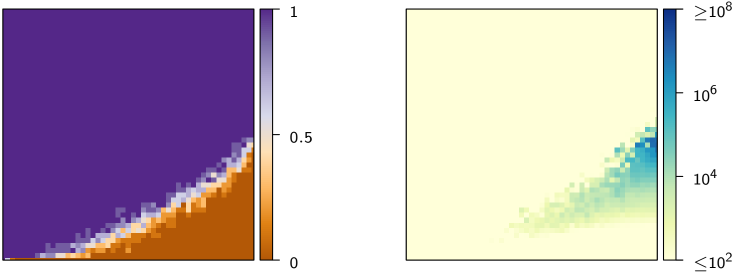
Generating Hard Subgraph Isomorphism Instances

- We can solve some random problem instances with a thousand pattern vertices, and ten thousand target vertices. Can we solve *any* instance with these sizes?
- We like having lots of instances, to make sure we don't overfit algorithm parameters.
- How do we randomly create subgraph isomorphism instances?

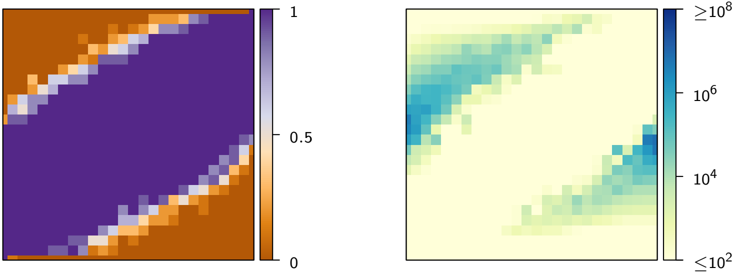
Phase Transitions



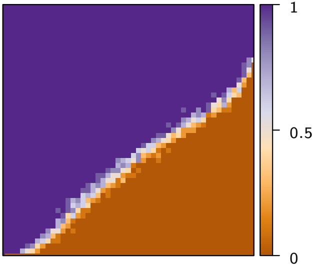
Phase Transitions

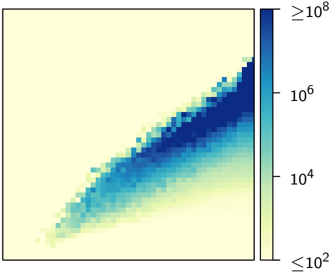


Phase Transitions

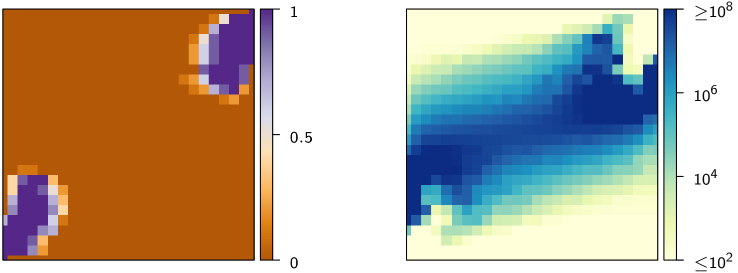


Phase Transitions





Phase Transitions



Graph Algorithms and Optimisation

- How do we solve problems that are “subgraph isomorphism plus some other constraints”?

http://dcs.gla.ac.uk/~ciaran
c.mccreesh.1@research.gla.ac.uk