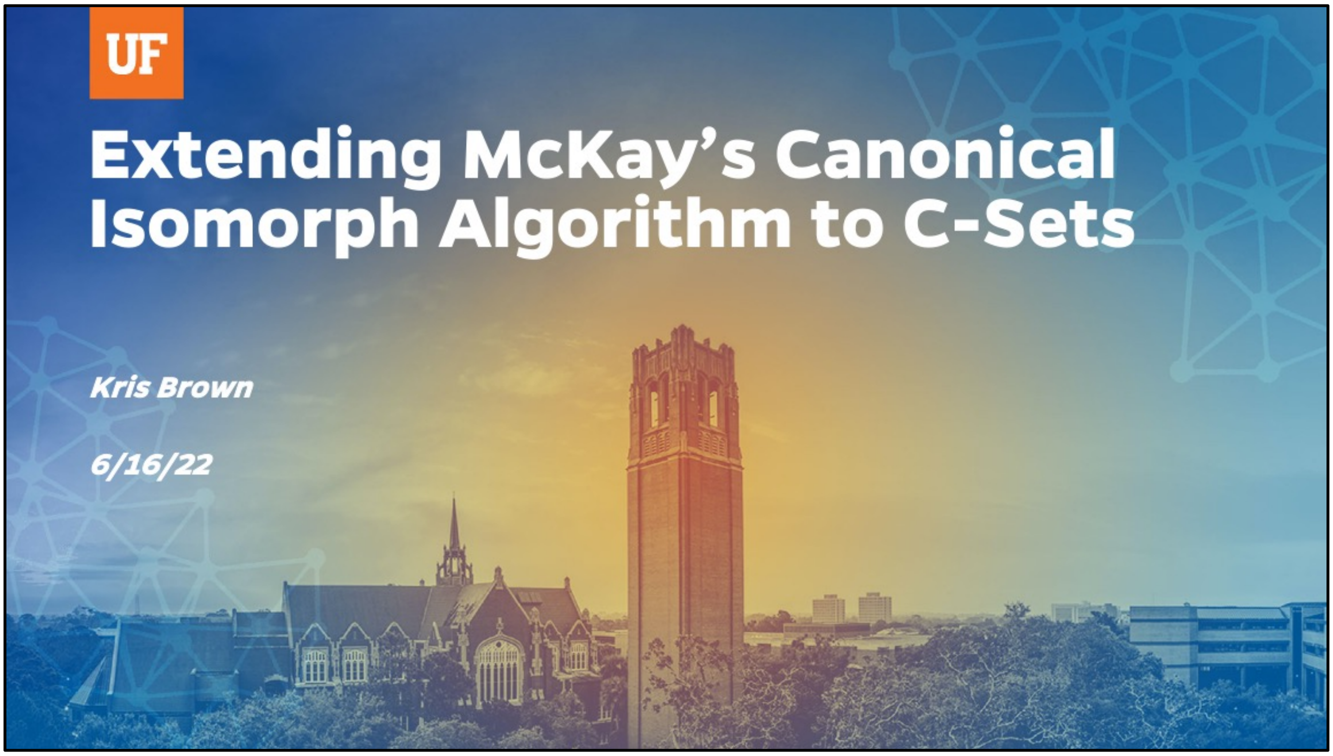


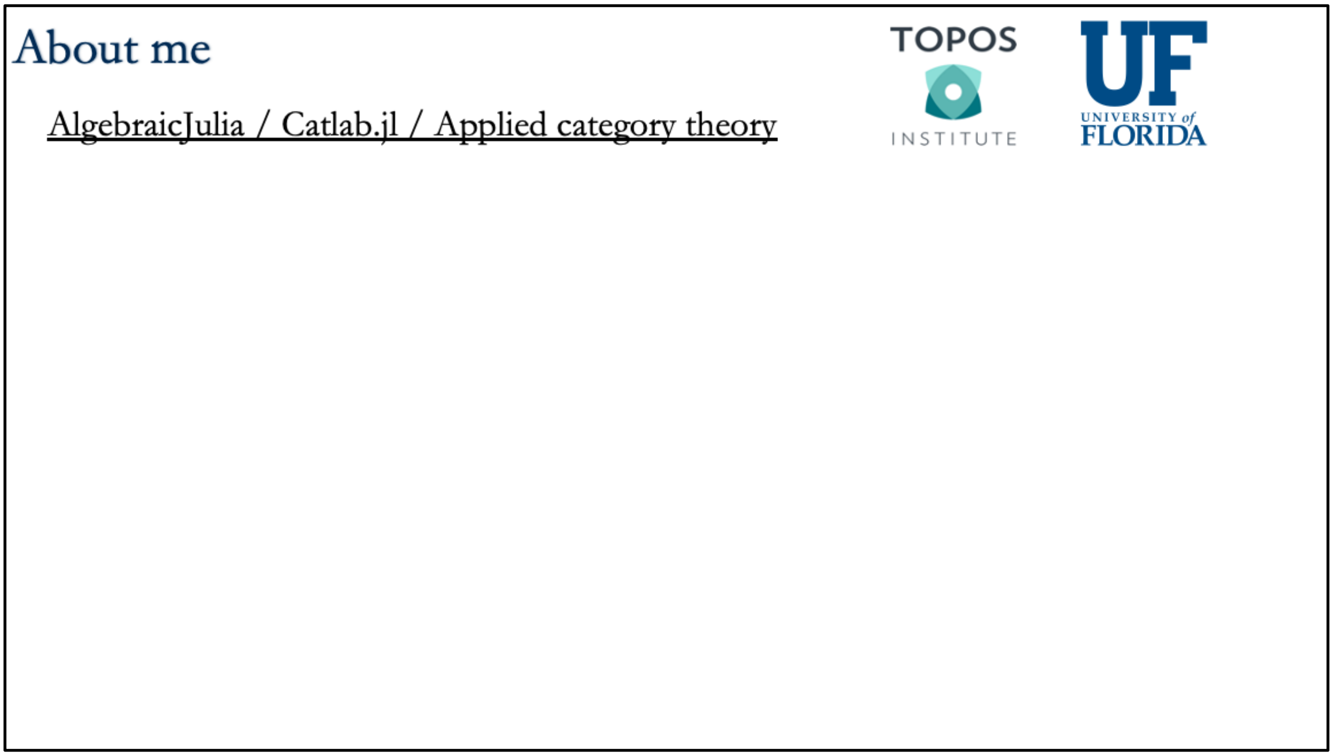
UF

Extending McKay's Canonical Isomorph Algorithm to C-Sets

Kris Brown

6/16/22

Hi, I want to talk about this useful algorithm which computes a canonical representative of a graph's isomorphism class and show a small generalization to the data structures we frequently use in our research group.

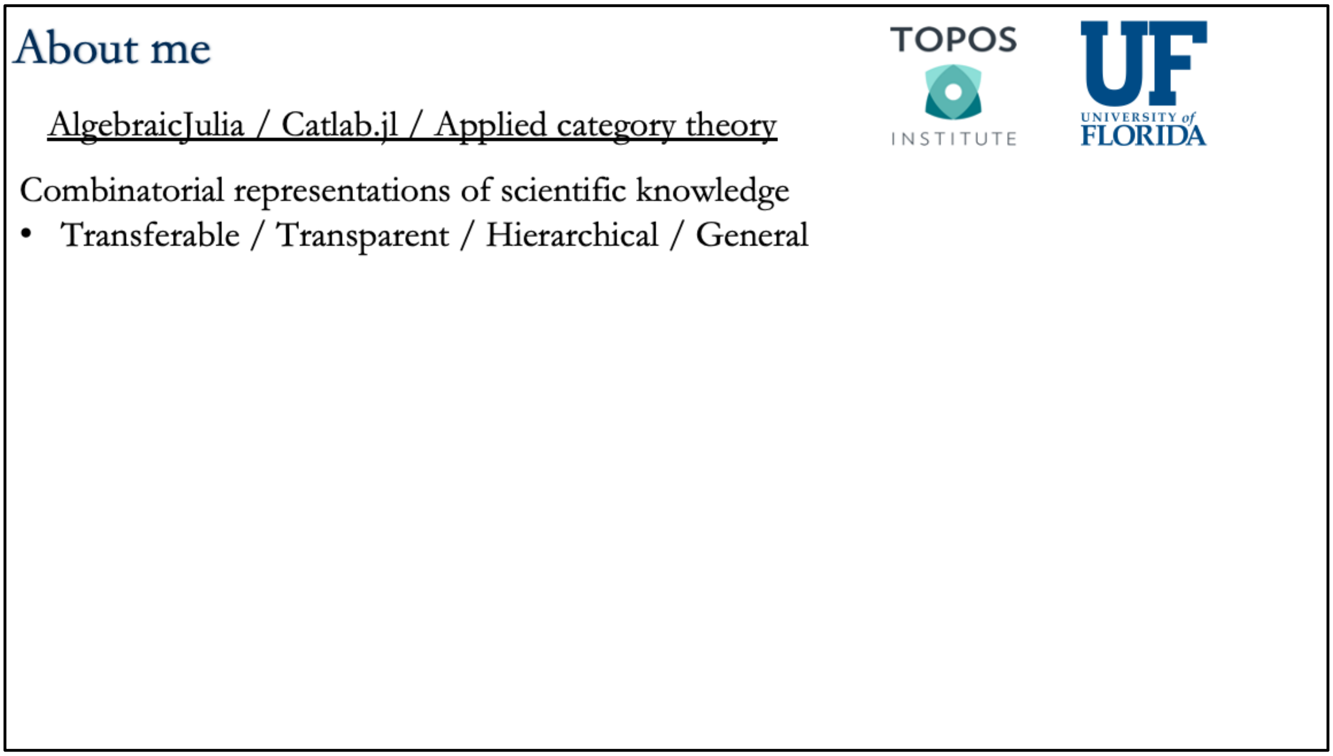


About me

AlgebraicJulia / Catlab.jl / Applied category theory

Brief background about me, since I’m a bit out of place here at a discrete math conference.

I’m a postdoc working with UF and the Topos Institute, where we have a team of applied mathematicians and engineers building a software ecosystem called AlgebraicJulia, where we use category theory and combinatorial data structures to solve problems in scientific computing.



About me

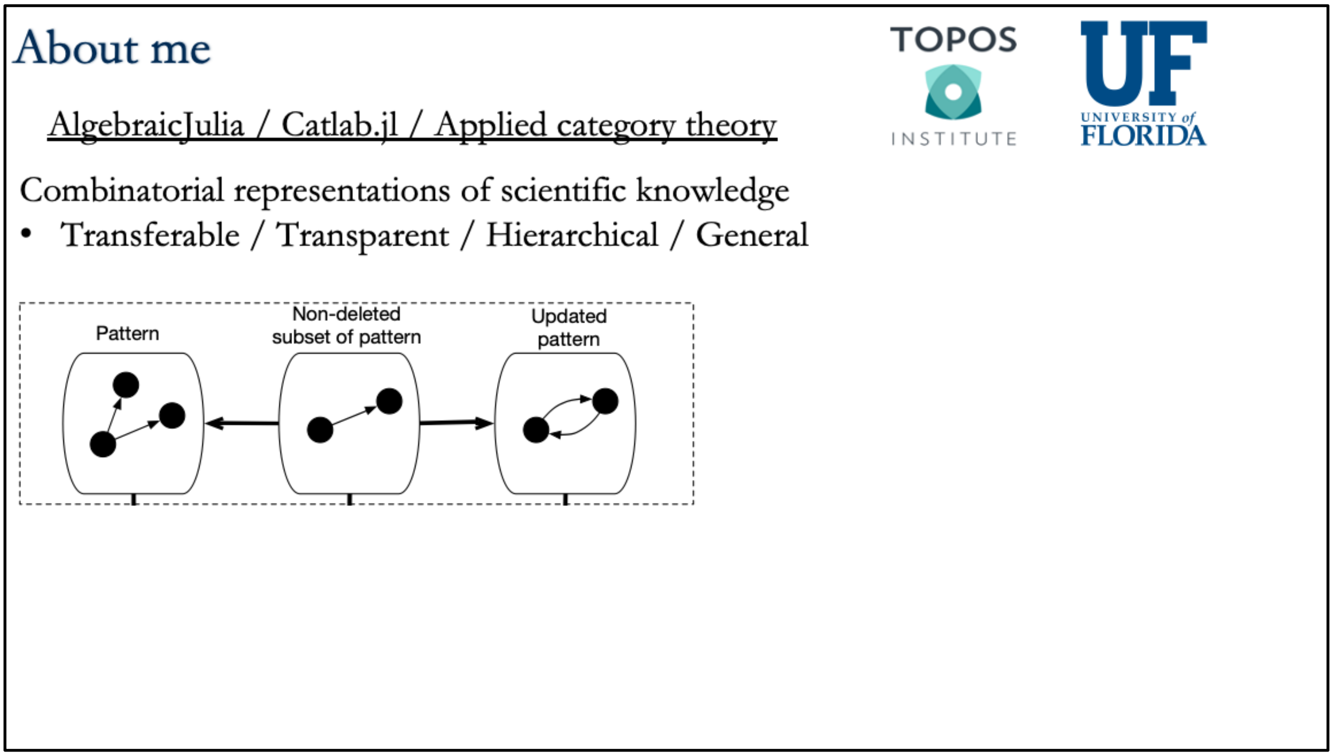
AlgebraicJulia / Catlab.jl / Applied category theory

Combinatorial representations of scientific knowledge

- Transferable / Transparent / Hierarchical / General

Scientific knowledge, when it's formalized at all which is rarely, often comes in the form of either mathematical expressions, formal logic, or code. These are all nice things, but our point of view is that a large fraction of the knowledge can have its syntax and semantics separated, where a lot of the knowledge can be encoded in a syntax of combinatorial data structures.

This has some broad advantages, because it's a lot easier to reason about combinatorial data structures than it is to perform analysis on arbitrary code/logic/mathmatical expressions, in particular, the knowledge can be automatically transfered, is transparent to analysis, can be more hierarchically composable, and can be much more general, because you can easily vary the syntax and semantics independently and gain a lot of mileage from that.



About me

TOPOS INSTITUTE

UF UNIVERSITY of FLORIDA

AlgebraicJulia / Catlab.jl / Applied category theory

Combinatorial representations of scientific knowledge

- Transferable / Transparent / Hierarchical / General

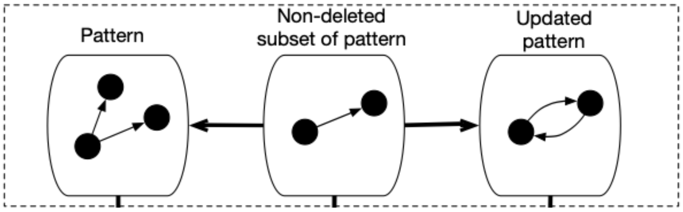
I have some examples on this slide, there are many more in a recent talk I gave on Youtube at the Topos institute

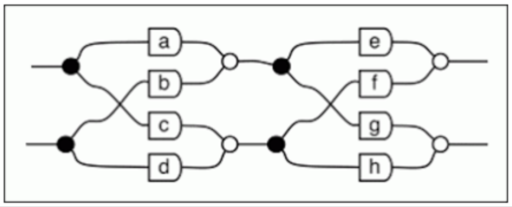
About me

AlgebraicJulia / Catlab.jl / Applied category theory

Combinatorial representations of scientific knowledge

- Transferable / Transparent / Hierarchical / General



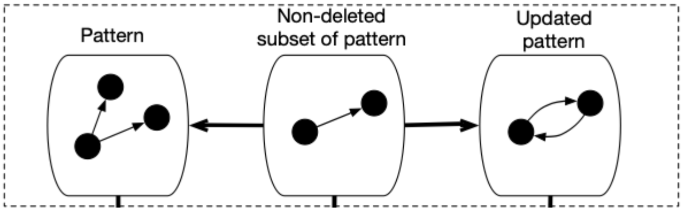


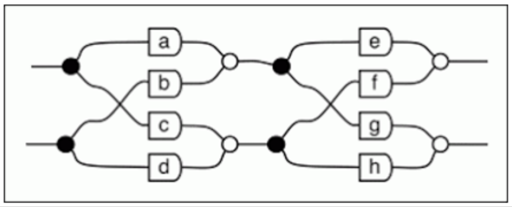
About me

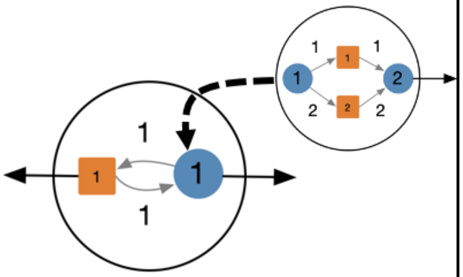
AlgebraicJulia / Catlab.jl / Applied category theory

Combinatorial representations of scientific knowledge

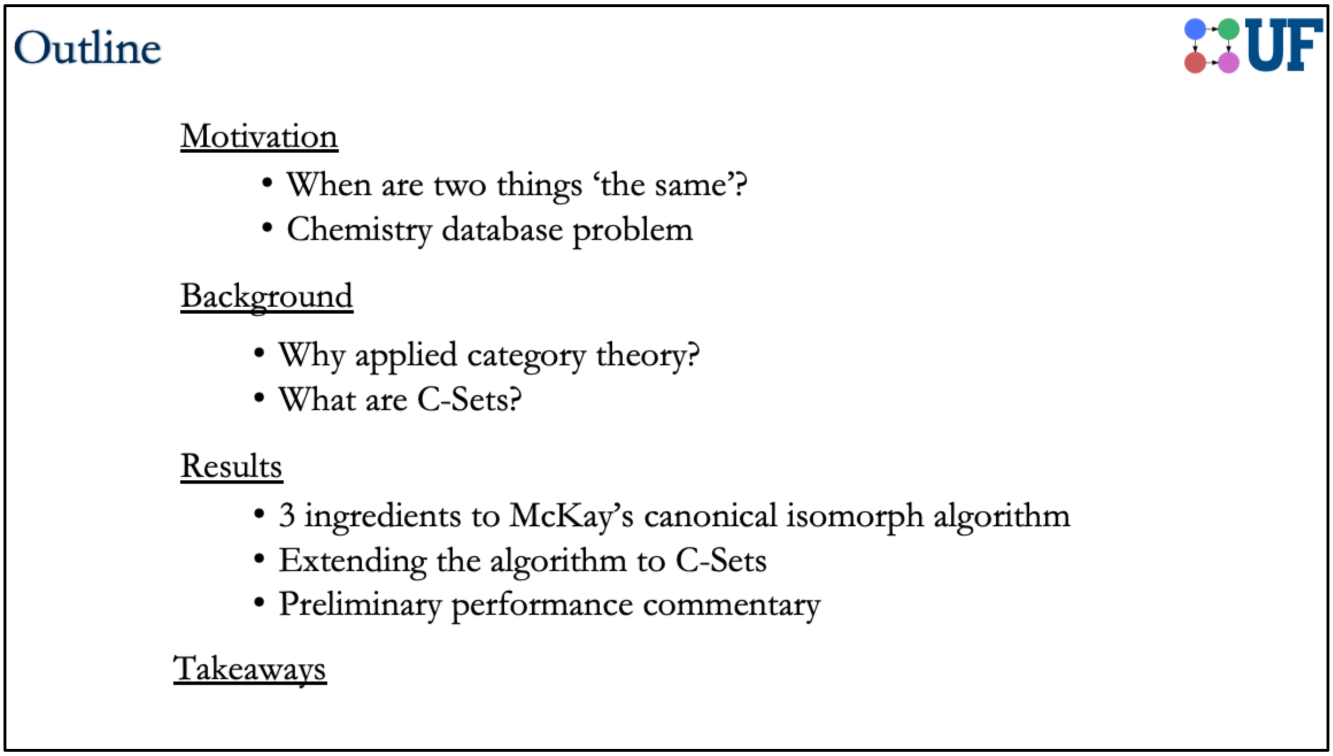
- Transferable / Transparent / Hierarchical / General







```
SELECT tran1.id, state1.id
FROM   S AS state1, T AS tran1,
       I AS in1,    O AS out1
WHERE  in1.is  = state1.id,
       in1.it  = tran1.id
       out1.os = state1.id
       out1.ot = tran1.id
```



Outline

Motivation

- When are two things 'the same'?
- Chemistry database problem

Background

- Why applied category theory?
- What are C-Sets?

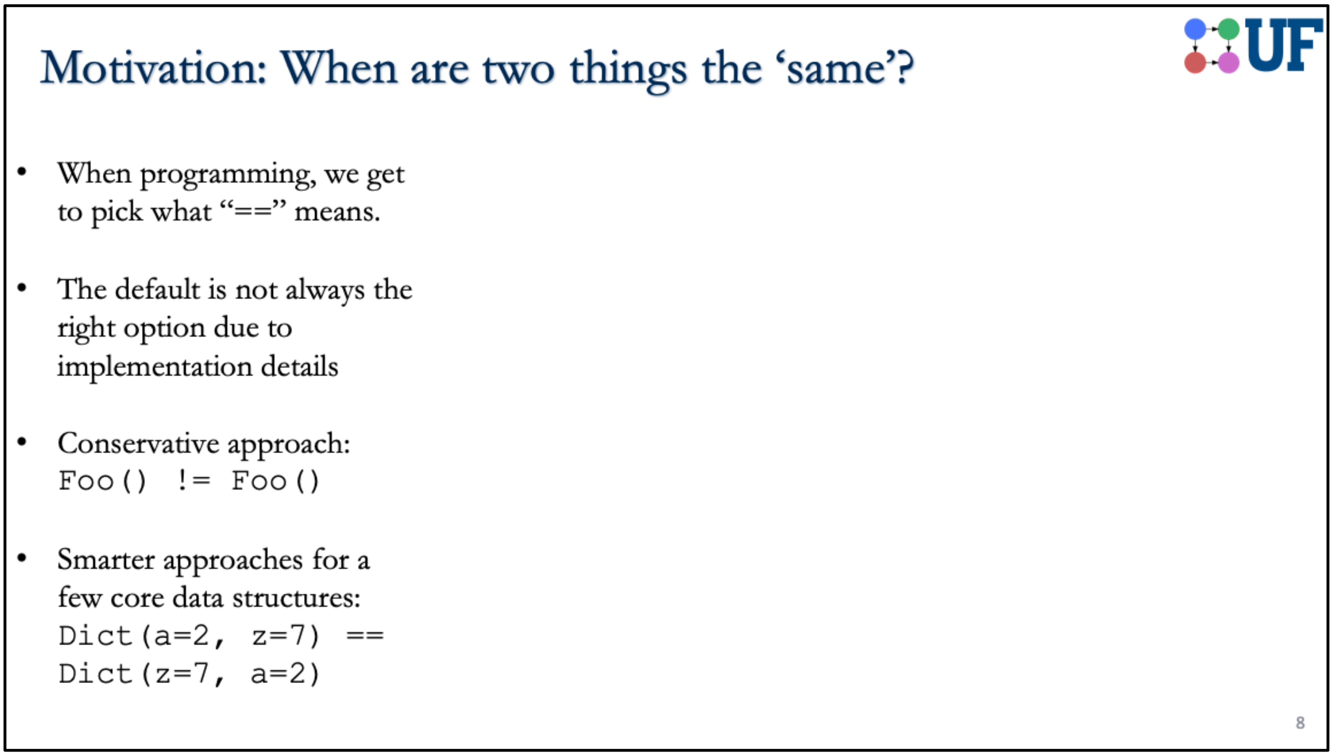
Results

- 3 ingredients to McKay's canonical isomorph algorithm
- Extending the algorithm to C-Sets
- Preliminary performance commentary

Takeaways

So this is the plan, to motivate why we want to generalize McKay's popular graph algorithm, I'll introduce the new level of generality and walk through the main pieces of it.

This is quite new work so a lot of what I have to say about performance is preliminary or conjectured, but I want to touch upon that as well as some more conceptual takeaways.



Motivation: When are two things the ‘same’?

- When programming, we get to pick what “==” means.
- The default is not always the right option due to implementation details
- Conservative approach:
  `Foo() != Foo()`
- Smarter approaches for a few core data structures:
  `Dict(a=2, z=7) ==`
  `Dict(z=7, a=2)`

8

So the problem I want to solve is prompted by this question of when two things are the ‘same’.

Some things are “literally” not the same but, in a certain context, we wish to treat them as the same.

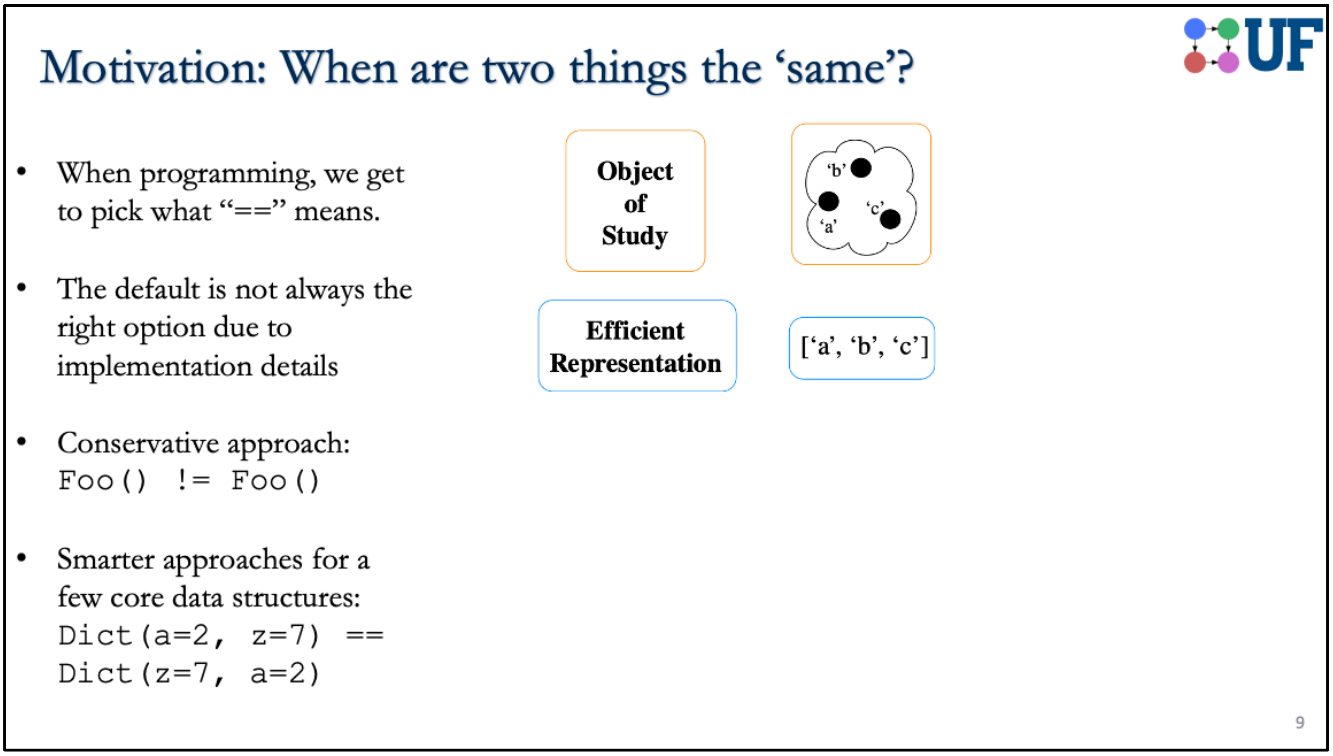
While I think there are lots of interesting real life and philosophical examples of this, I want to jump right to the where this shows up in programming.*CLICK*

There we often define datatypes and have the option of defining what procedure we want to run to test whether two instances of that type are equal.*CLICK*

There’s usually some default option, but it might not be right. *CLICK*

For example, often it just checks equality of reference for a user defined datatype.*CLICK*

For its built in data types, it can do smarter things.*CLICK*



Motivation: When are two things the 'same'?

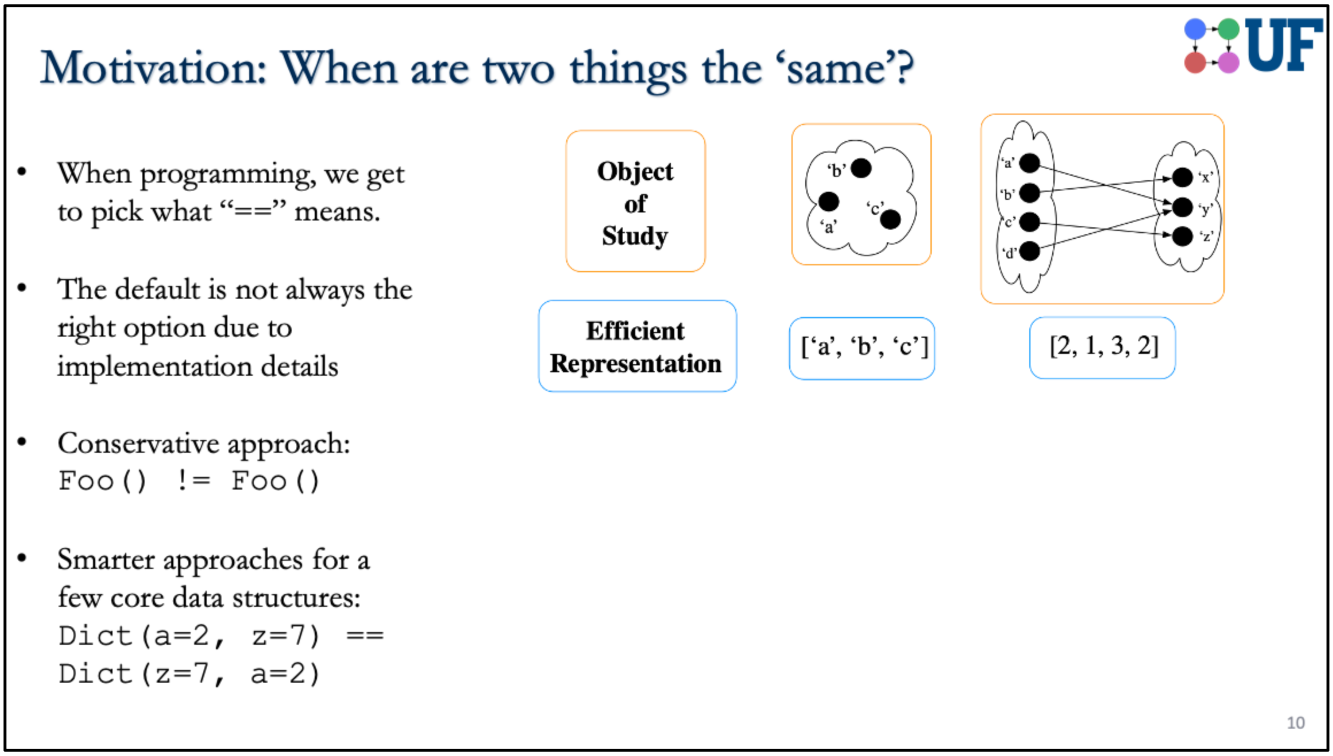
- When programming, we get to pick what "==" means.
- The default is not always the right option due to implementation details
- Conservative approach: `Foo() != Foo()`
- Smarter approaches for a few core data structures: `Dict(a=2, z=7) == Dict(z=7, a=2)`

Object of Study

Efficient Representation

['a', 'b', 'c']

When we start trying to represent mathematical objects in code efficiently, we find that the literal equality is not what we want. For example, I may want to represent this set as a vector. This is quite a bad thing to do considering that it imposes an order on elements which truly have no order, and it doesn't enforce there being no duplicates, so why would we do this?



Motivation: When are two things the 'same'?

- When programming, we get to pick what "==" means.
- The default is not always the right option due to implementation details
- Conservative approach: `Foo() != Foo()`
- Smarter approaches for a few core data structures: `Dict(a=2, z=7) == Dict(z=7, a=2)`

Object of Study

Efficient Representation

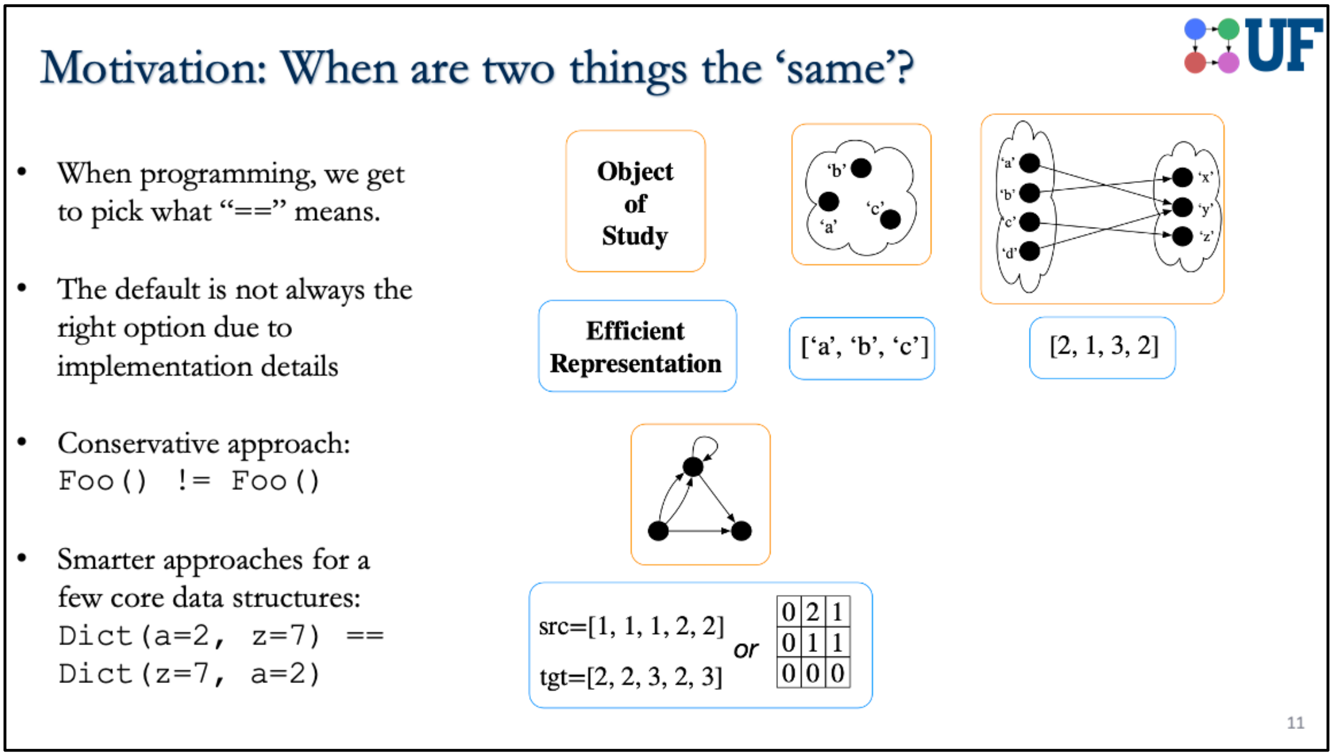
['a', 'b', 'c']

[2, 1, 3, 2]

'b'
'c'
'a'
'a'
'b'
'c'
'd'
'x'
'y'
'z'

10

It allows me to represent FUNCTIONS between sets very efficiently. For each element of the domain, the corresponding vector element can say which element of the codomain it's sent to. So, again, set theoretically, it's the same set if it's ['b','c','a'] or ['a','a','b','c'], but you'd have to tell your programming language that you want to sort and remove duplicates before checking equality, for example.



Motivation: When are two things the ‘same’?

- When programming, we get to pick what “==” means.
- The default is not always the right option due to implementation details
- Conservative approach: `Foo() != Foo()`
- Smarter approaches for a few core data structures: `Dict(a=2, z=7) == Dict(z=7, a=2)`

Object of Study

Efficient Representation

[‘a’, ‘b’, ‘c’]

[2, 1, 3, 2]

src=[1, 1, 1, 2, 2]

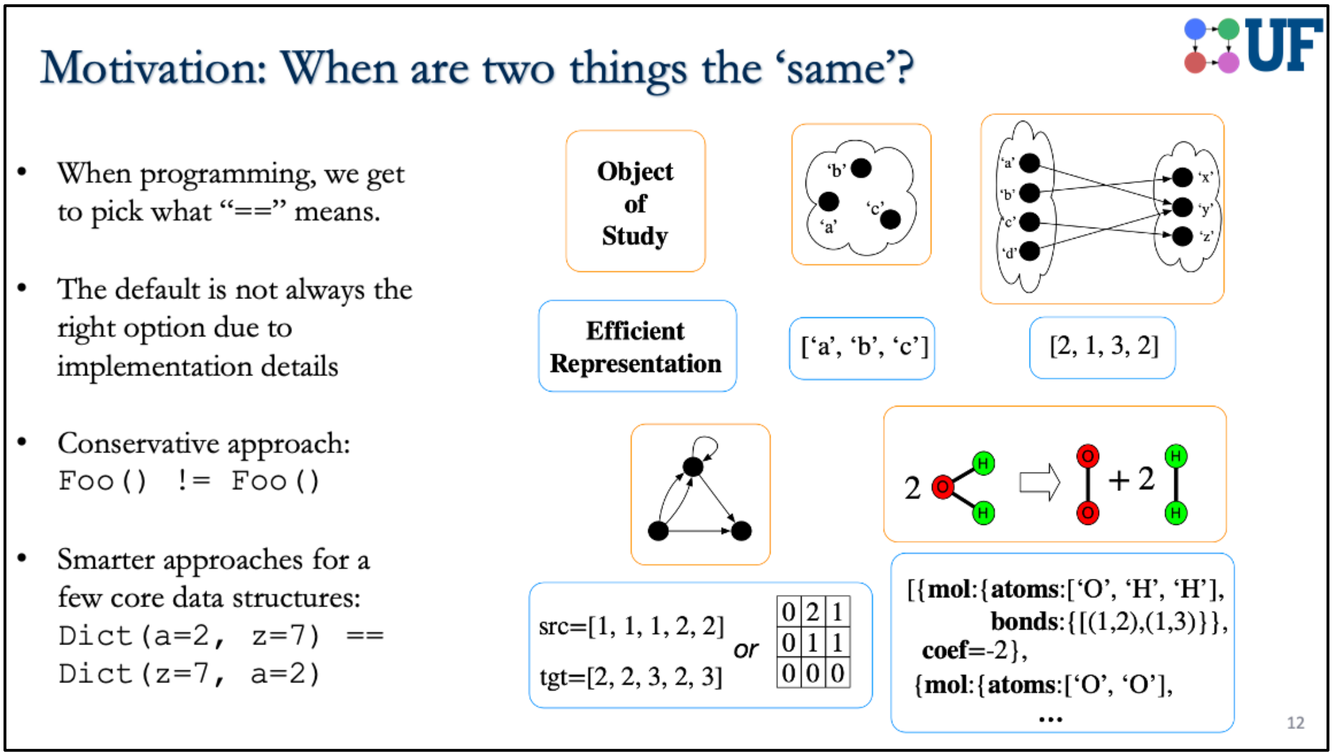
tgt=[2, 2, 3, 2, 3]

or

0	2	1
0	1	1
0	0	0

‘b’
‘c’
‘a’
‘a’
‘b’
‘c’
‘d’
‘x’
‘y’
‘z’

Graphs also have a few ways we can efficiently represent. But if it’s no longer the same graph if we reorder the vertices or the edges, that is at odds with how we think of the graph. If I drew this graph differently and shifted positions of vertices, it'd be the same graph for the purposes of a graph theorist, but not an artist. So that’s what’s interesting: the context is really important! It’s also interesting that here math is really being prescriptive, rather than merely descriptive, in virtue of you choosing to refer to this thing on your computer with the graph theorist’s word.



Motivation: When are two things the 'same'?

- When programming, we get to pick what "==" means.
- The default is not always the right option due to implementation details
- Conservative approach: `Foo() != Foo()`
- Smarter approaches for a few core data structures: `Dict(a=2, z=7) == Dict(z=7, a=2)`

Object of Study

Efficient Representation

['a', 'b', 'c']

[2, 1, 3, 2]

src=[1, 1, 1, 2, 2]
tgt=[2, 2, 3, 2, 3]

or

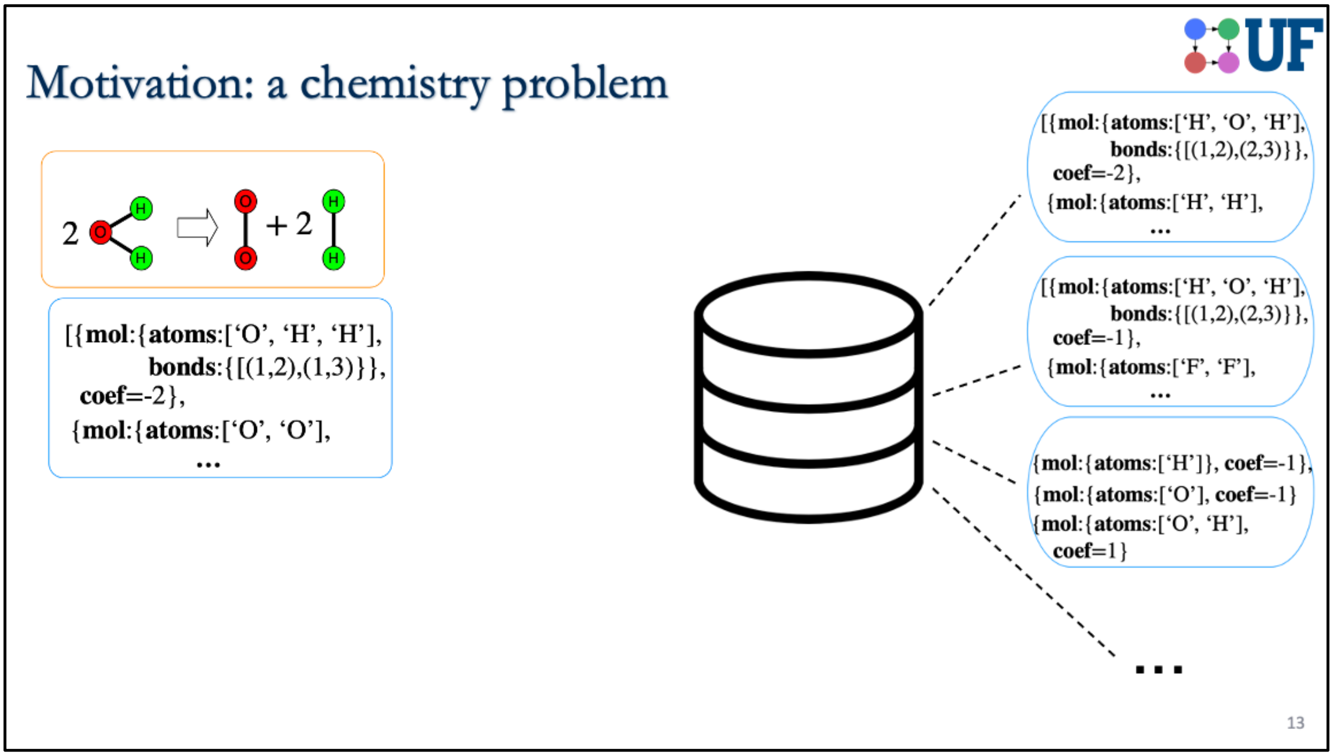
$$\begin{array}{|c|c|c|}\hline 0 & 2 & 1 \\ \hline 0 & 1 & 1 \\ \hline 0 & 0 & 0 \\ \hline\end{array}$$

[{**mol**:{**atoms**:['O', 'H', 'H'],
bonds:{[(1,2),(1,3)]}},
coef=-2},
{**mol**:{**atoms**:['O', 'O'],
...

These are all very standard objects of study in math/cs, but what about a scientist who has some custom data structure representing their domain of interest?

We want it to be natural to declare datatype along with the context of its use such that a non-computer scientist will automatically get the right notion of equality.

Our running example will be a chemical reaction, such as this one that says two water molecules react to produce oxygen and two hydrogen diatomic molecules. A chemist's natural representation of this using built-in datatypes is also shown below.



Motivation: a chemistry problem

```
[{mol:{atoms:[‘O’, ‘H’, ‘H’],
        bonds:{[(1,2),(1,3)}},
  coef=-2},
 {mol:{atoms:[‘O’, ‘O’],
      …
```

```
[{mol:{atoms:[‘H’, ‘O’, ‘H’],
        bonds:{[(1,2),(2,3)}},
  coef=-2},
 {mol:{atoms:[‘H’, ‘H’],
      …
```

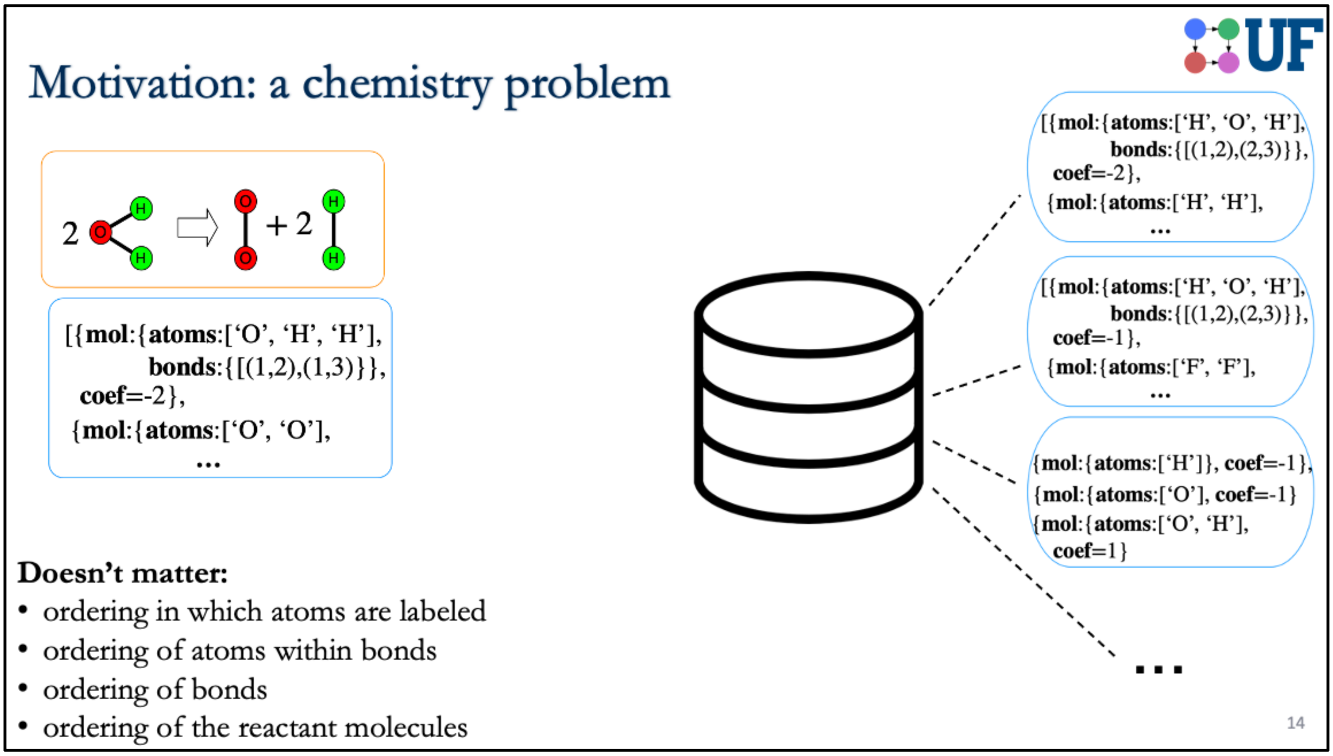
```
[{mol:{atoms:[‘H’, ‘O’, ‘H’],
        bonds:{[(1,2),(2,3)}},
  coef=-1},
 {mol:{atoms:[‘F’, ‘F’],
      …
```

```
{mol:{atoms:[‘H’]}, coef=-1},
{mol:{atoms:[‘O’], coef=-1}
{mol:{atoms:[‘O’, ‘H’],
  coef=1}
```

…

Our working example problem is the following: we have a particular reaction of interest and a database with millions of such reactions.

We want to test if the reaction is in the database, but this not as simple as testing the literal equality of our data structure



Motivation: a chemistry problem

[{**mol**:{**atoms**:['O', 'H', 'H'],
bonds:{[(1,2),(1,3)]}},
coef=-2},
{**mol**:{**atoms**:['O', 'O'],
...

[{**mol**:{**atoms**:['H', 'O', 'H'],
bonds:{[(1,2),(2,3)]}},
coef=-2},
{**mol**:{**atoms**:['H', 'H'],
...

[{**mol**:{**atoms**:['H', 'O', 'H'],
bonds:{[(1,2),(2,3)]}},
coef=-1},
{**mol**:{**atoms**:['F', 'F'],
...

{**mol**:{**atoms**:['H']}, **coef**=-1},
{**mol**:{**atoms**:['O'], **coef**=-1}
{**mol**:{**atoms**:['O', 'H'],
coef=1}

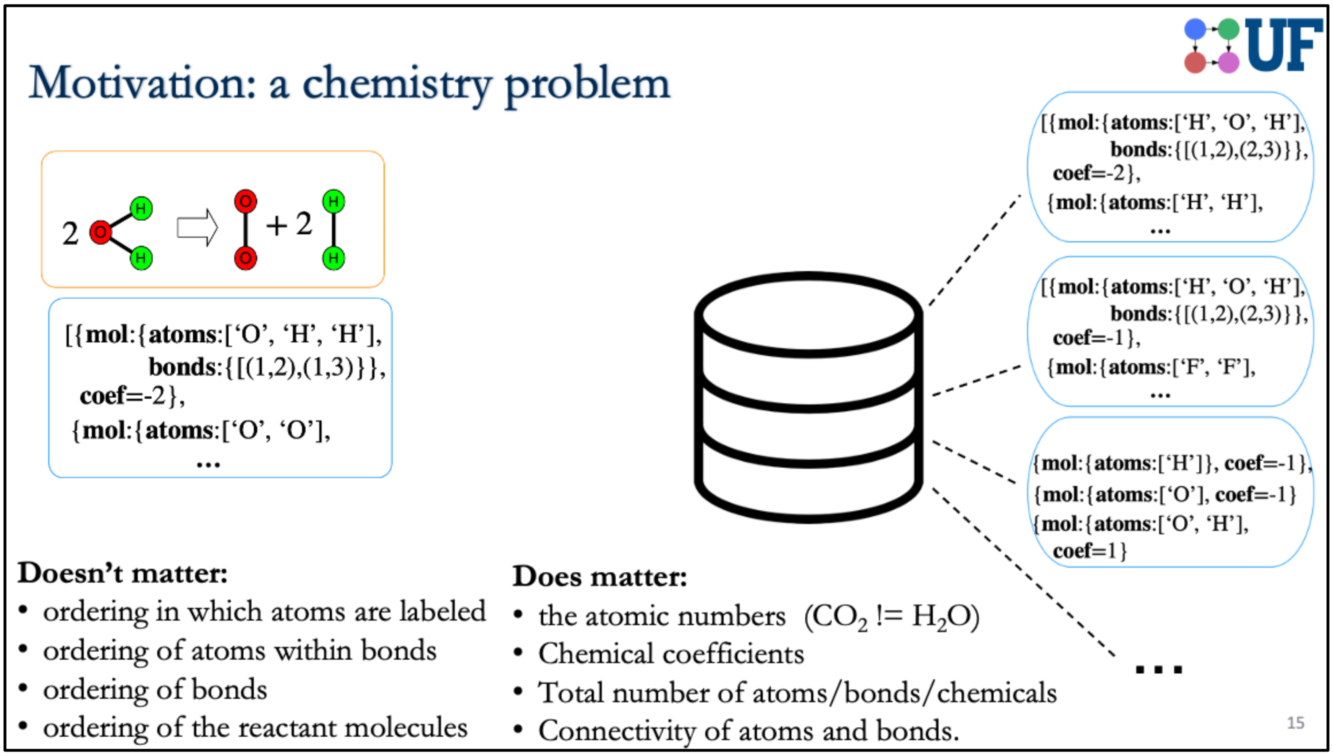
...

Doesn't matter:

- ordering in which atoms are labeled
- ordering of atoms within bonds
- ordering of bonds
- ordering of the reactant molecules

14

We need to be flexible with respect to all sorts of permutations.



Motivation: a chemistry problem

```
[{mol:{atoms:[‘O’, ‘H’, ‘H’],
        bonds:{[(1,2),(1,3)]}},
  coef=-2},
 {mol:{atoms:[‘O’, ‘O’],
       …
```

```
[{mol:{atoms:[‘H’, ‘O’, ‘H’],
        bonds:{[(1,2),(2,3)]}},
  coef=-2},
 {mol:{atoms:[‘H’, ‘H’],
       …
```

```
[{mol:{atoms:[‘H’, ‘O’, ‘H’],
        bonds:{[(1,2),(2,3)]}},
  coef=-1},
 {mol:{atoms:[‘F’, ‘F’],
       …
```

```
{mol:{atoms:[‘H’]}, coef=-1},
{mol:{atoms:[‘O’], coef=-1}
{mol:{atoms:[‘O’, ‘H’],
  coef=1}
```

…

Doesn’t matter:

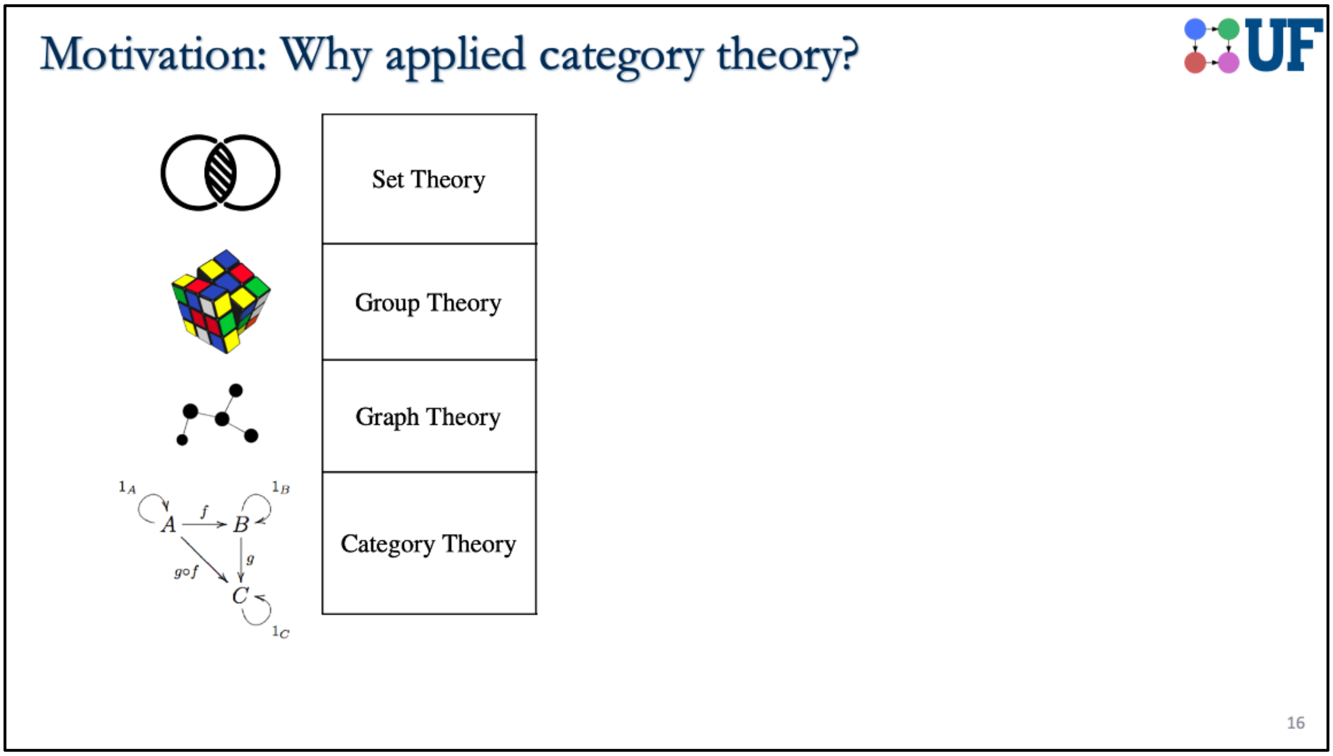
- ordering in which atoms are labeled
- ordering of atoms within bonds
- ordering of bonds
- ordering of the reactant molecules

Does matter:

- the atomic numbers (CO_2 != H_2O)
- Chemical coefficients
- Total number of atoms/bonds/chemicals
- Connectivity of atoms and bonds.

15

But there is important information that must be satisfied for our query to consider, most importantly the connectivity of atoms and bottoms.



Motivation: Why applied category theory?

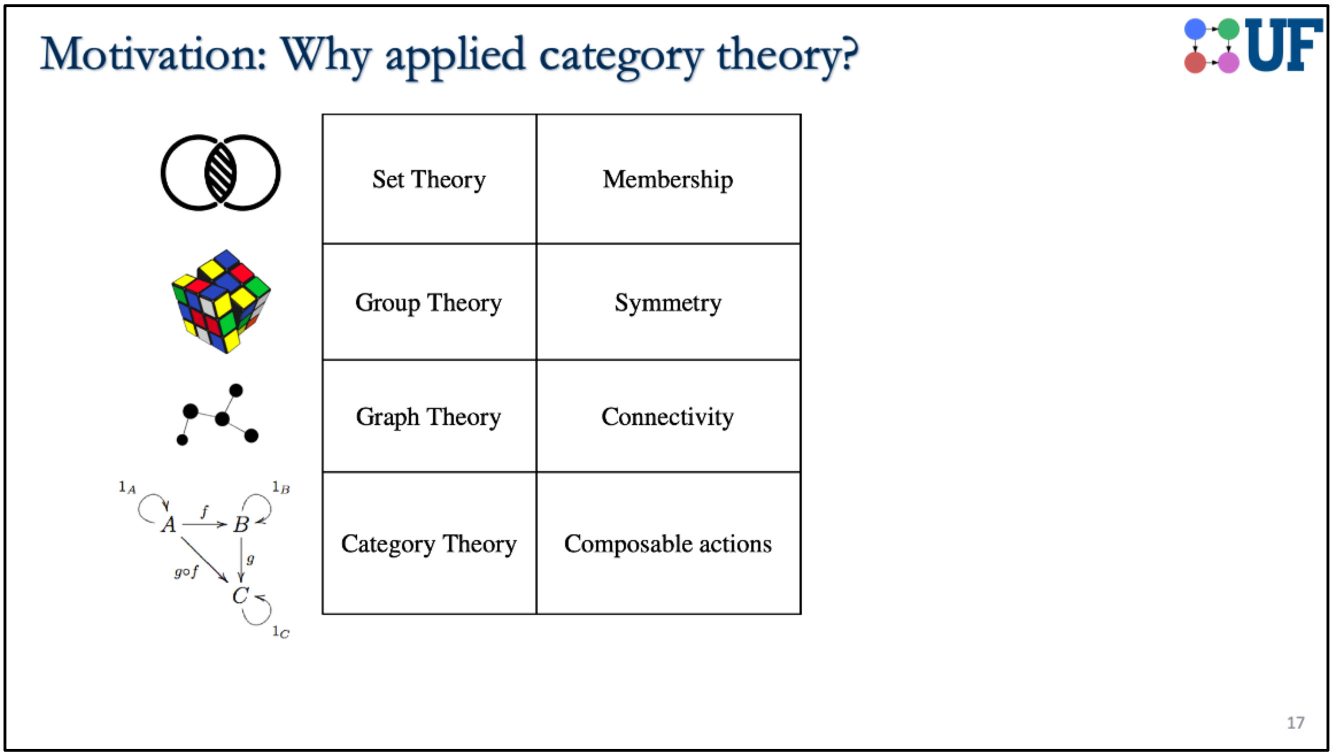
Set Theory
Group Theory
Graph Theory
Category Theory

$$\begin{tikzcd} A \arrow[loop left, "1_A"] \arrow[r, "f"] \arrow[dr, "g\circ f"'] & B \arrow[loop right, "1_B"] \arrow[d, "g"] \\ & C \arrow[loop right, "1_C"] \end{tikzcd}$$

So how do we tackle this problem in great generality, such that we can handle things as diverse as graphs and chemical reactions as special cases?

Applied category theory is very useful here. If you’re not familiar with category theory, I’ll try to introduce it without getting bogged down in details, because we are really only going to use some of its most basic ingredients.

First: how to think of category theory in relation to other branches of math you may be more familiar with?



Motivation: Why applied category theory?

$$\begin{tikzcd} A \arrow[loop above, "1_A"] \arrow[r, "f"] \arrow[dr, "g\circ f"'] & B \arrow[loop above, "1_B"] \arrow[d, "g"] \\ & C \arrow[loop below, "1_C"] \end{tikzcd}$$

Set Theory	Membership
Group Theory	Symmetry
Graph Theory	Connectivity
Category Theory	Composable actions

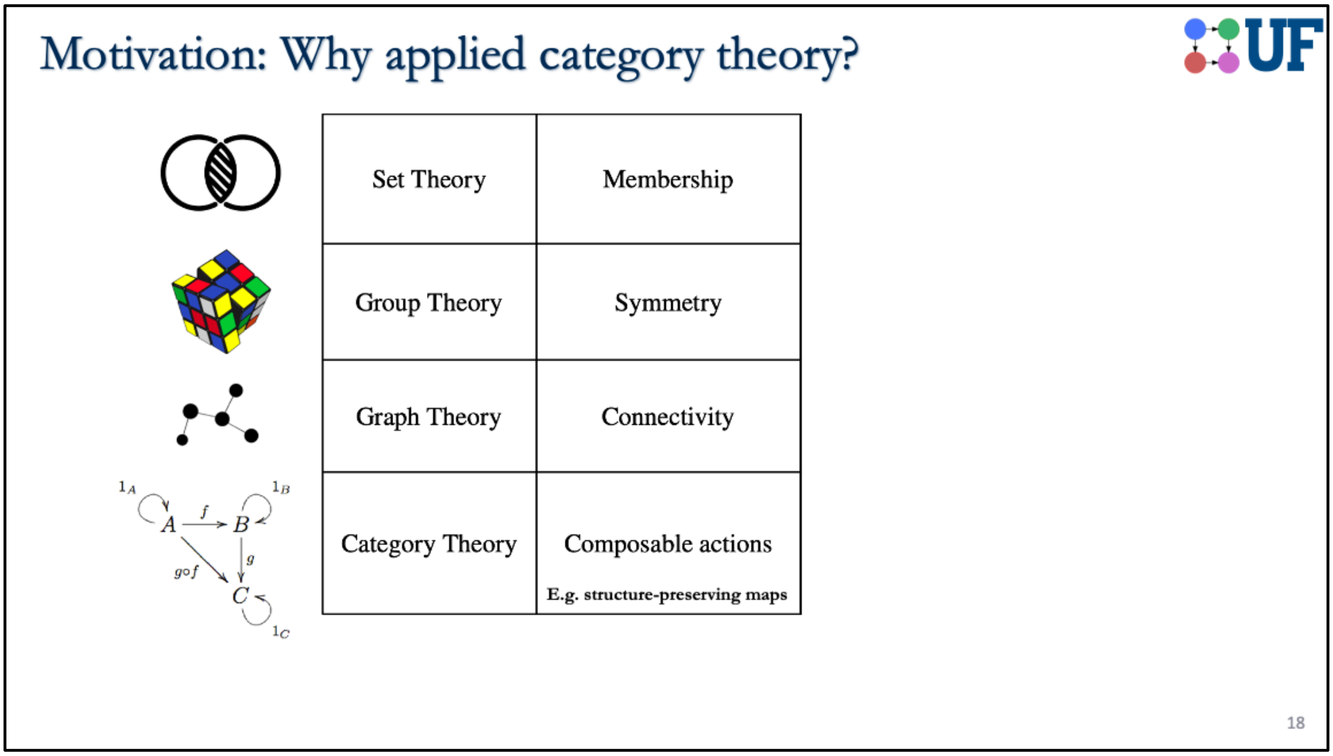
17

You can think of set theory as axiomatizing some object that gives us the minimum language needed to talk about membership.

Likewise axiomatizing a group can be thought of as giving us the minimum language needed to talk about symmetry, thus group theory becomes the study of symmetries.

Graphs are a minimum language to talk about connectivity.

And so, by analogy, I could introduce category theory as the minimum language that lets us talk about composible processes.



Motivation: Why applied category theory?

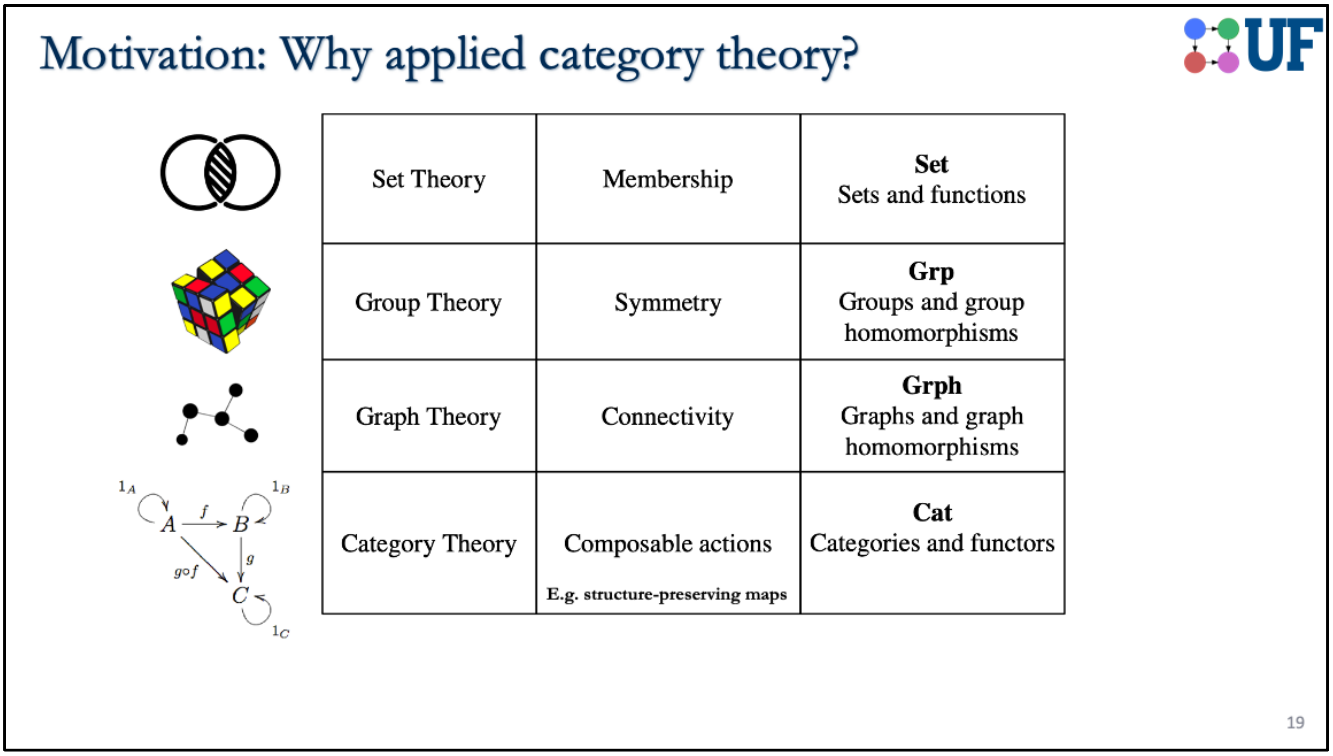
Set Theory	Membership
Group Theory	Symmetry
Graph Theory	Connectivity
Category Theory	Composable actions E.g. structure-preserving maps

$$\begin{tikzcd} A \arrow[loop left, "1_A"] \arrow[r, "f"] \arrow[dr, "g\circ f"'] & B \arrow[loop right, "1_B"] \arrow[d, "g"] \\ & C \arrow[loop right, "1_C"] \end{tikzcd}$$

18

One really important choice for what those composable actions are would be structure-preserving maps.

If you study some domain purely by looking at the structure-preserving maps, you end up at the domain structurally. This is why another characterization of category theory is that it is the study of ‘structure’. Given that other branches of math concern themselves with studying the structure of their object of interest (e.g. group theorists are interested in the Klein 4 group, but they don’t particular care what four elements are in the group’s set), this allows some to think of category theory as the simplest language which lets us model other branches of mathematics.



Motivation: Why applied category theory?

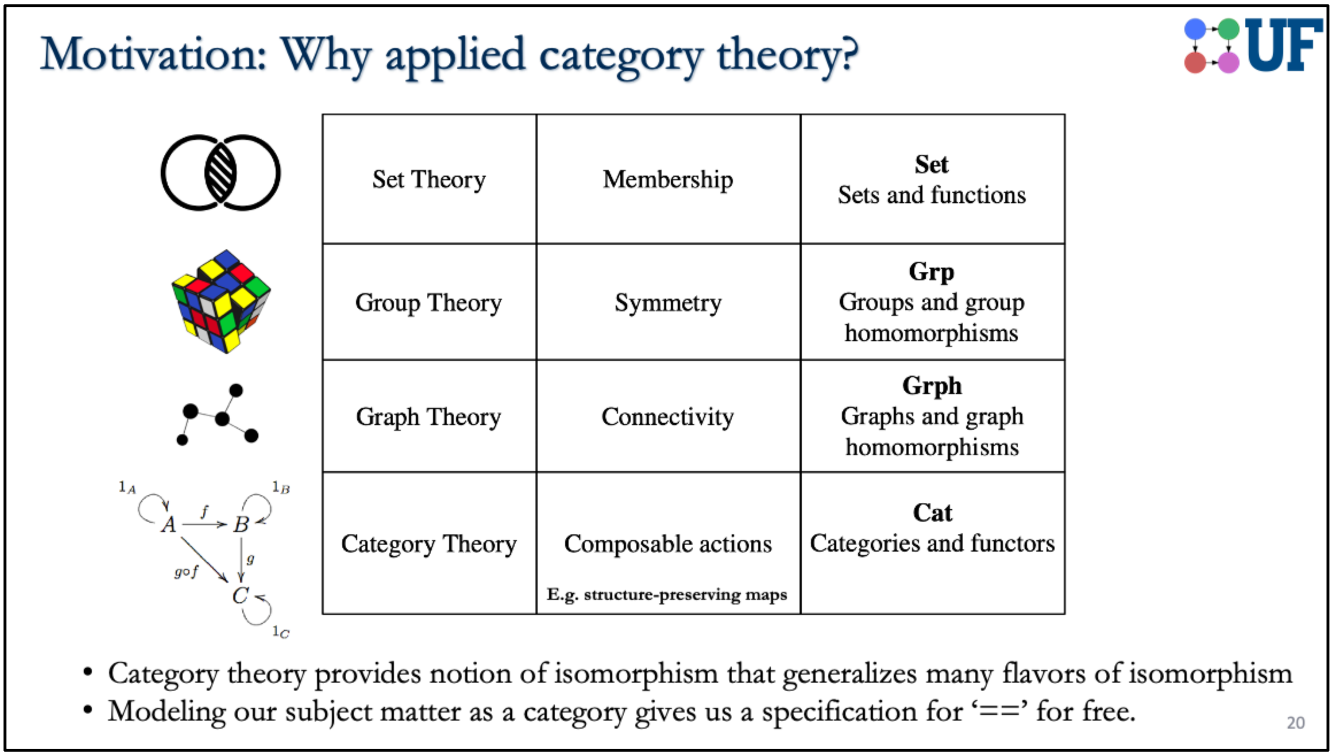
Set Theory	Membership	**Set** Sets and functions
Group Theory	Symmetry	**Grp** Groups and group homomorphisms
Graph Theory	Connectivity	**Grph** Graphs and graph homomorphisms
Category Theory	Composable actions E.g. structure-preserving maps	**Cat** Categories and functors

$$\begin{tikzcd} A \arrow[loop left, "1_A"] \arrow[r, "f"] \arrow[dr, "g\circ f"'] & B \arrow[loop right, "1_B"] \arrow[d, "g"] \\ & C \arrow[loop right, "1_C"] \end{tikzcd}$$

19

In fact, each of these branches of math can be itself viewed as a category with structure preserving maps.

So a lot of cool things come about from being able to connect all these disciplines, such as realizing that the Cartesian product of sets is really the same product as multiplying numbers, or the product of groups, or the greatest common divisor etc.



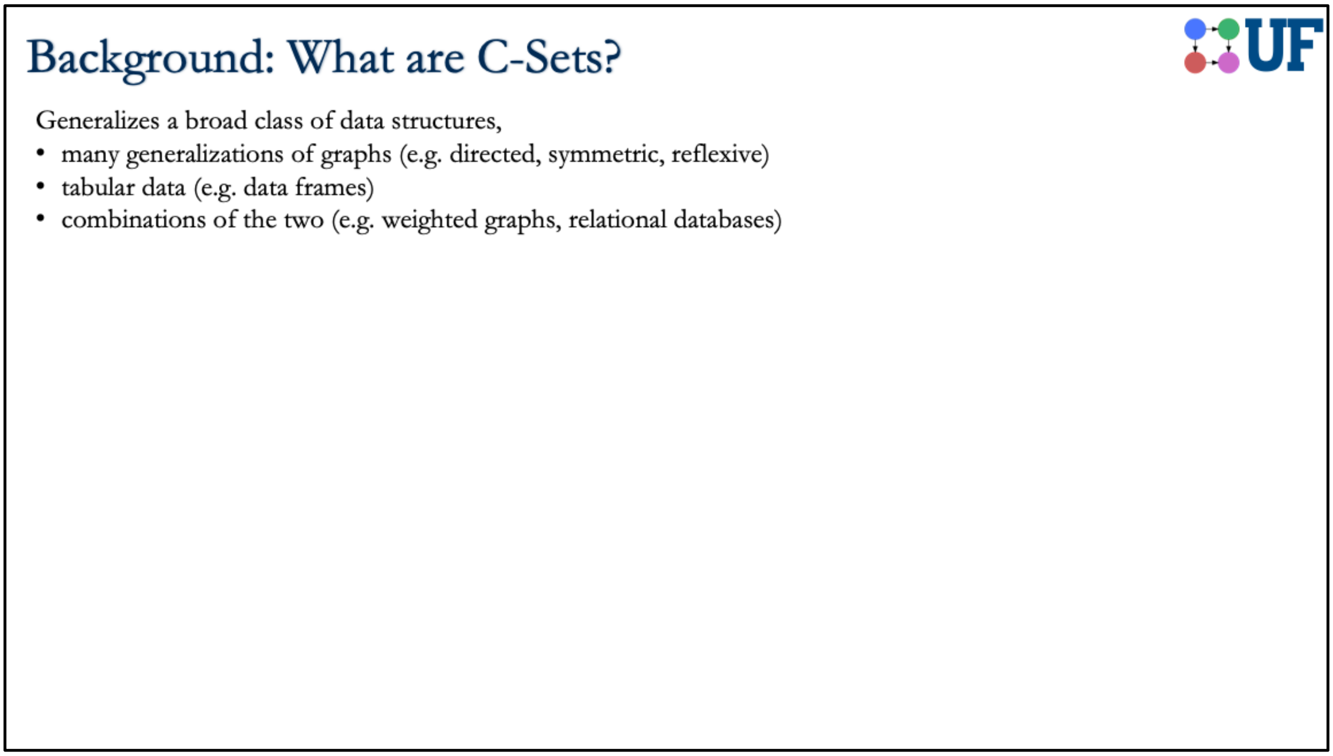
Motivation: Why applied category theory?

Set Theory	Membership	**Set** Sets and functions
Group Theory	Symmetry	**Grp** Groups and group homomorphisms
Graph Theory	Connectivity	**Grph** Graphs and graph homomorphisms
Category Theory	Composable actions **E.g. structure-preserving maps**	**Cat** Categories and functors

$$\begin{tikzcd} A \arrow[loop left, "1_A"] \arrow[r, "f"] \arrow[dr, "g\circ f"'] & B \arrow[loop right, "1_B"] \arrow[d, "g"] \\ & C \arrow[loop below, "1_C"] \end{tikzcd}$$

- Category theory provides notion of isomorphism that generalizes many flavors of isomorphism
- Modeling our subject matter as a category gives us a specification for '==' for free.

Because category theory tells us what the right notion of isomorphism is for a given category, if we model our domain as a category then we get a notion of sameness for free.

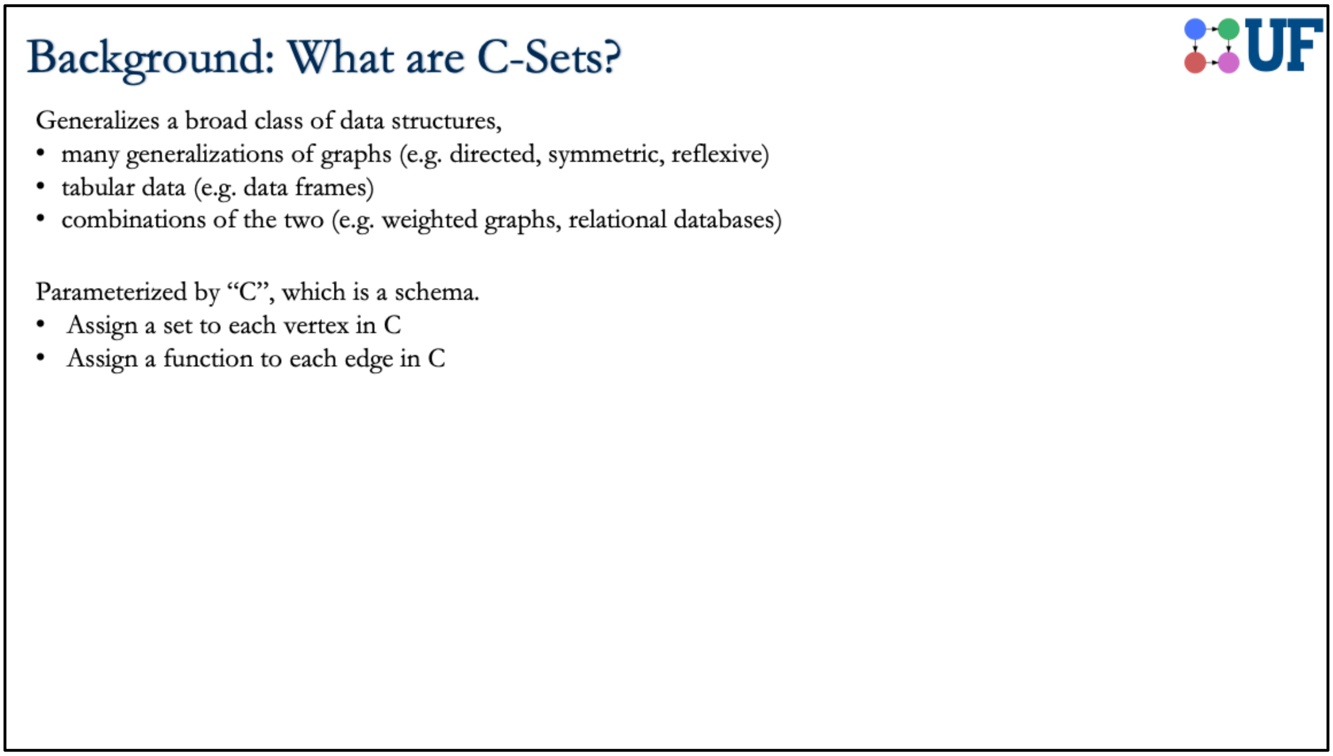


Background: What are C-Sets?

Generalizes a broad class of data structures,
- many generalizations of graphs (e.g. directed, symmetric, reflexive)
- tabular data (e.g. data frames)
- combinations of the two (e.g. weighted graphs, relational databases)

I’m going to introduce C-sets, which are a particular type of category which have a well-defined, computable notion of equivalence but also are incredibly expressive, enough to handle most basic scientific and engineering models.

They generalize many flavors of graphs, tabular data, as well as combinations of the two. If you’re not familiar with relational databases, they have been the de facto standard of data modeling in professional applications over the past forty years.



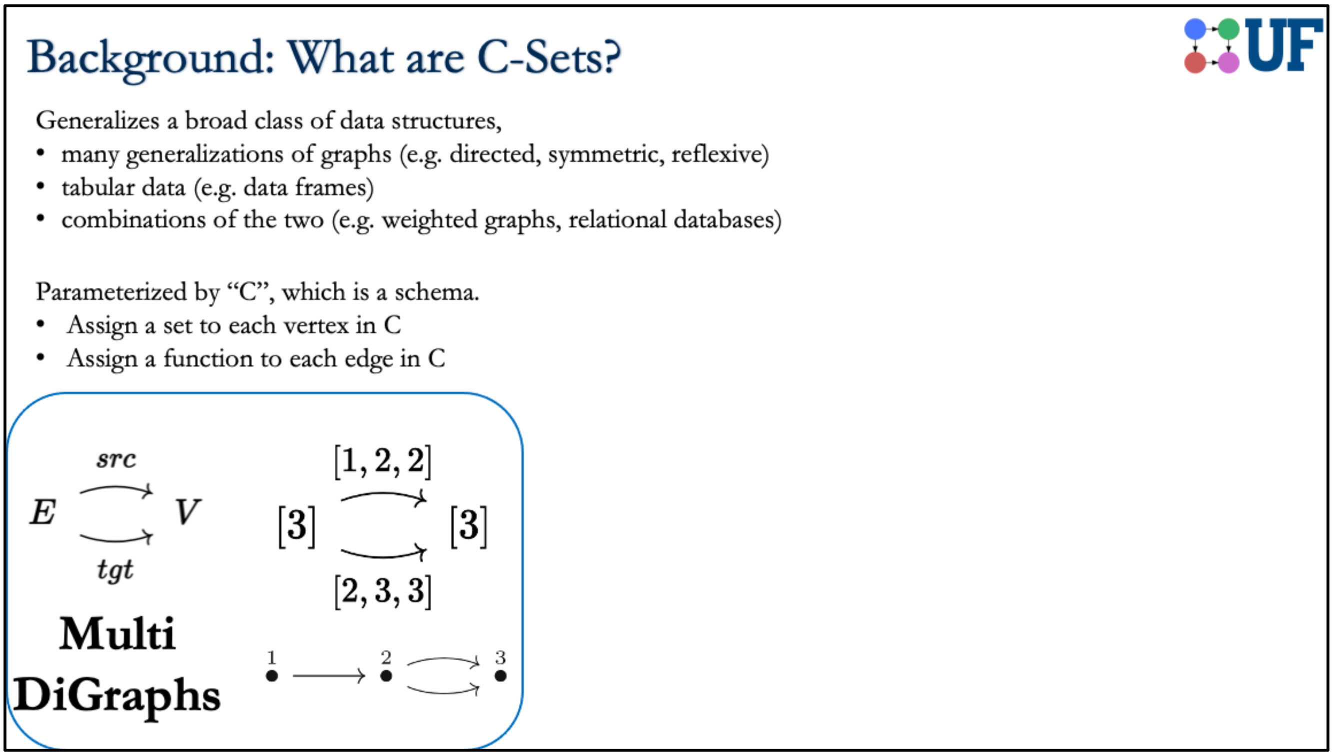
Background: What are C-Sets?

Generalizes a broad class of data structures,
- many generalizations of graphs (e.g. directed, symmetric, reflexive)
- tabular data (e.g. data frames)
- combinations of the two (e.g. weighted graphs, relational databases)

Parameterized by "C", which is a schema.
- Assign a set to each vertex in C
- Assign a function to each edge in C

A C-Set isn't a specific data type, but rather once you provide a "C", you get a data type. This "C" plays the role of a schema for a relational database.

The category theoretic definition of a C-set is a functor from C (which is a category) to **Set**. But all that means is that you assign a set for each vertex in C, a function for each edge in C, and you also satisfy equational laws that C has. Let me show some examples.



Background: What are C-Sets?

Generalizes a broad class of data structures,

- many generalizations of graphs (e.g. directed, symmetric, reflexive)
- tabular data (e.g. data frames)
- combinations of the two (e.g. weighted graphs, relational databases)

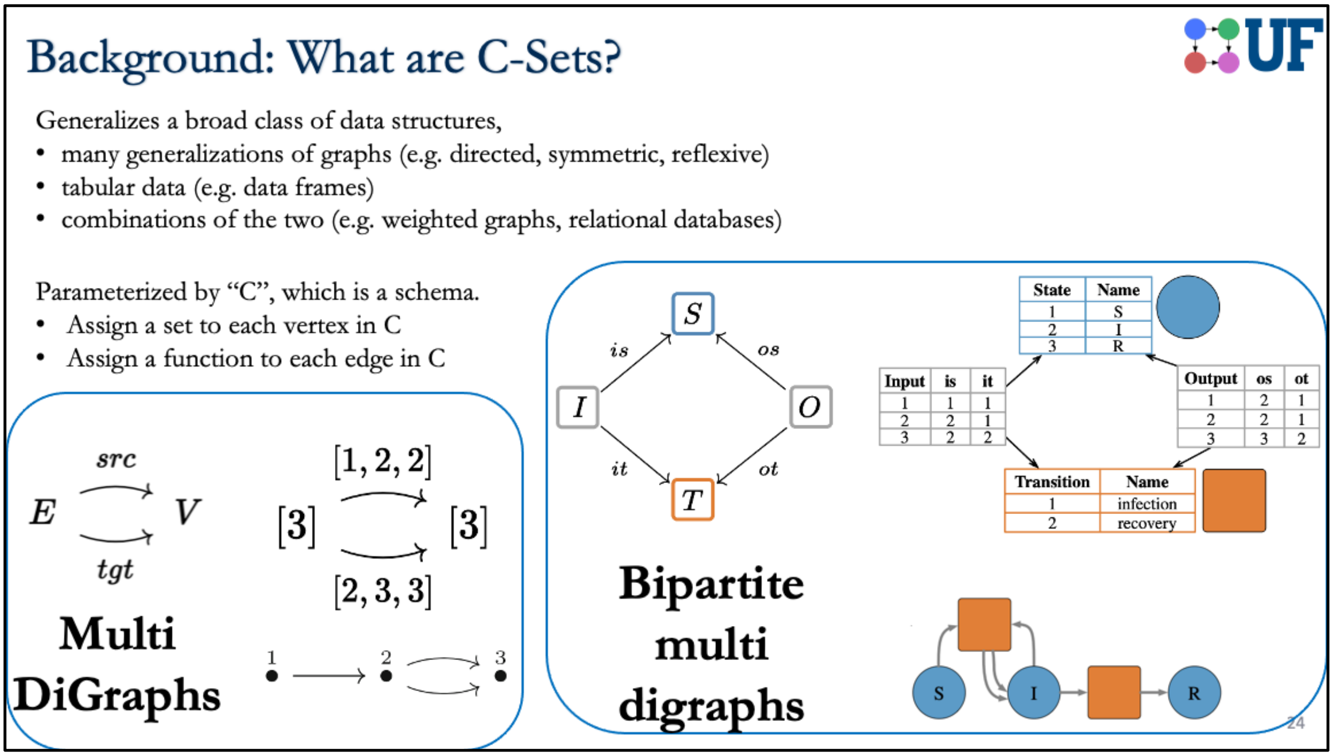
Parameterized by “C”, which is a schema.

- Assign a set to each vertex in C
- Assign a function to each edge in C

$$\begin{tikzcd} E \arrow[r, bend left, "src"] \arrow[r, bend right, "tgt"'] & V \end{tikzcd} \qquad \begin{tikzcd} {[3]} \arrow[r, bend left, "{[1,2,2]}"] \arrow[r, bend right, "{[2,3,3]}"'] & {[3]} \end{tikzcd}$$

Multi DiGraphs

Here on the top left is the schema for graphs. If you plug this in as “C”, a C-set contains the data of a graph. Note we are using that strategy of representing sets as just vectors [1,2,3,...] and functions as vectors. If we want to represent this graph here at the bottom, we say that the edge and vertex sets have three elements, and these functions here define source and target.



Background: What are C-Sets?

Generalizes a broad class of data structures,

- many generalizations of graphs (e.g. directed, symmetric, reflexive)
- tabular data (e.g. data frames)
- combinations of the two (e.g. weighted graphs, relational databases)

Parameterized by "C", which is a schema.

- Assign a set to each vertex in C
- Assign a function to each edge in C

$$\begin{tikzcd} E \arrow[r, bend left, "src"] \arrow[r, bend right, "tgt"'] & V \end{tikzcd}$$

$$\begin{tikzcd} {[3]} \arrow[r, bend left, "{[1,2,2]}"] \arrow[r, bend right, "{[2,3,3]}"'] & {[3]} \end{tikzcd}$$

Multi DiGraphs

$$\begin{tikzcd} & S & \\ I \arrow[ur, "is"] \arrow[dr, "it"'] & & O \arrow[ul, "os"'] \arrow[dl, "ot"] \\ & T & \end{tikzcd}$$

State	Name
1	S
2	I
3	R

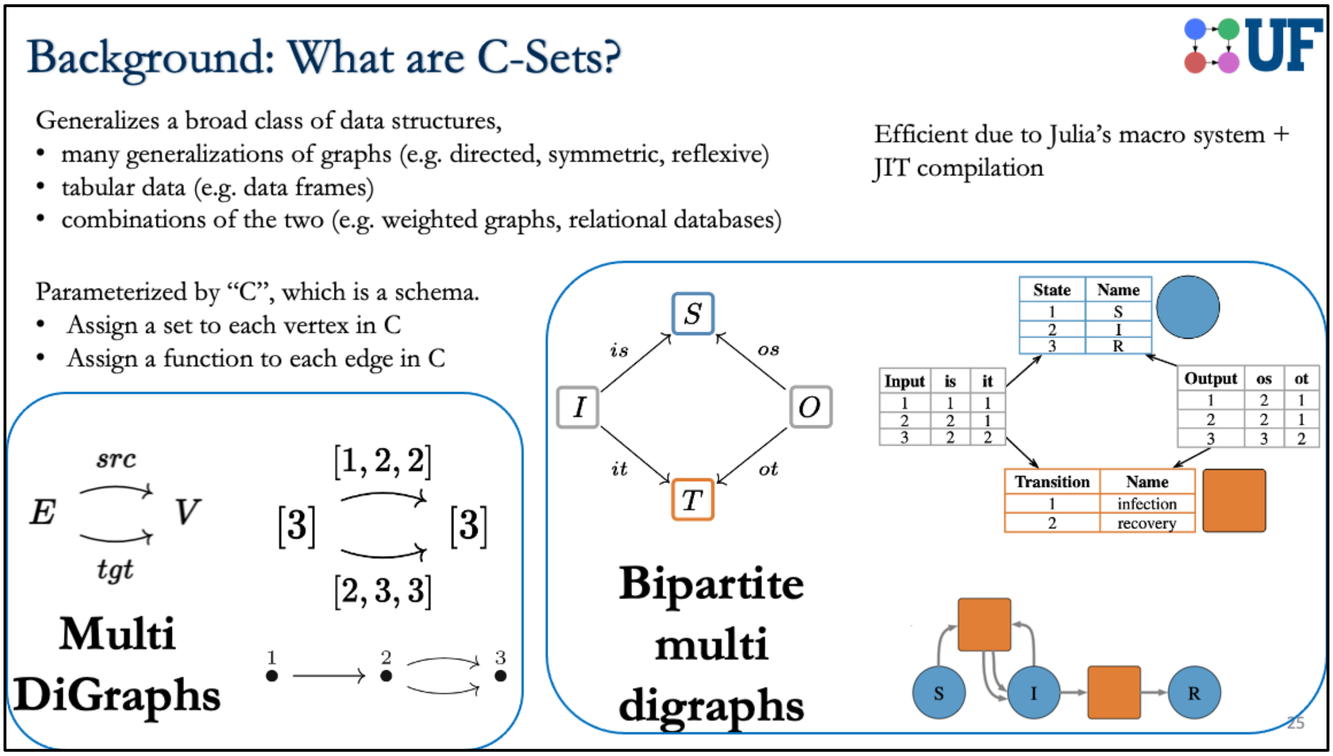
Input	is	it
1	1	1
2	2	1
3	2	2

Output	os	ot
1	2	1
2	2	1
3	3	2

Transition	Name
1	infection
2	recovery

Bipartite multi digraphs

Slightly more complicated is this schema for bipartite graphs. We visualize these by making one vertex type blue and the other orange, and we have two types of arrows each with their own source and target functions.



Background: What are C-Sets?

Generalizes a broad class of data structures,

- many generalizations of graphs (e.g. directed, symmetric, reflexive)
- tabular data (e.g. data frames)
- combinations of the two (e.g. weighted graphs, relational databases)

Efficient due to Julia's macro system + JIT compilation

Parameterized by "C", which is a schema.

- Assign a set to each vertex in C
- Assign a function to each edge in C

$$\begin{tikzcd} E \arrow[r, bend left, "src"] \arrow[r, bend right, "tgt"'] & V \end{tikzcd} \qquad \begin{tikzcd} {[3]} \arrow[r, bend left, "{[1,2,2]}"] \arrow[r, bend right, "{[2,3,3]}"'] & {[3]} \end{tikzcd}$$

Multi DiGraphs

$$\begin{tikzcd} & S & \\ I \arrow[ur, "is"] \arrow[dr, "it"'] & & O \arrow[ul, "os"'] \arrow[dl, "ot"] \\ & T & \end{tikzcd}$$

State	Name
1	S
2	I
3	R

Input	is	it
1	1	1
2	2	1
3	2	2

Output	os	ot
1	2	1
2	2	1
3	3	2

Transition	Name
1	infection
2	recovery

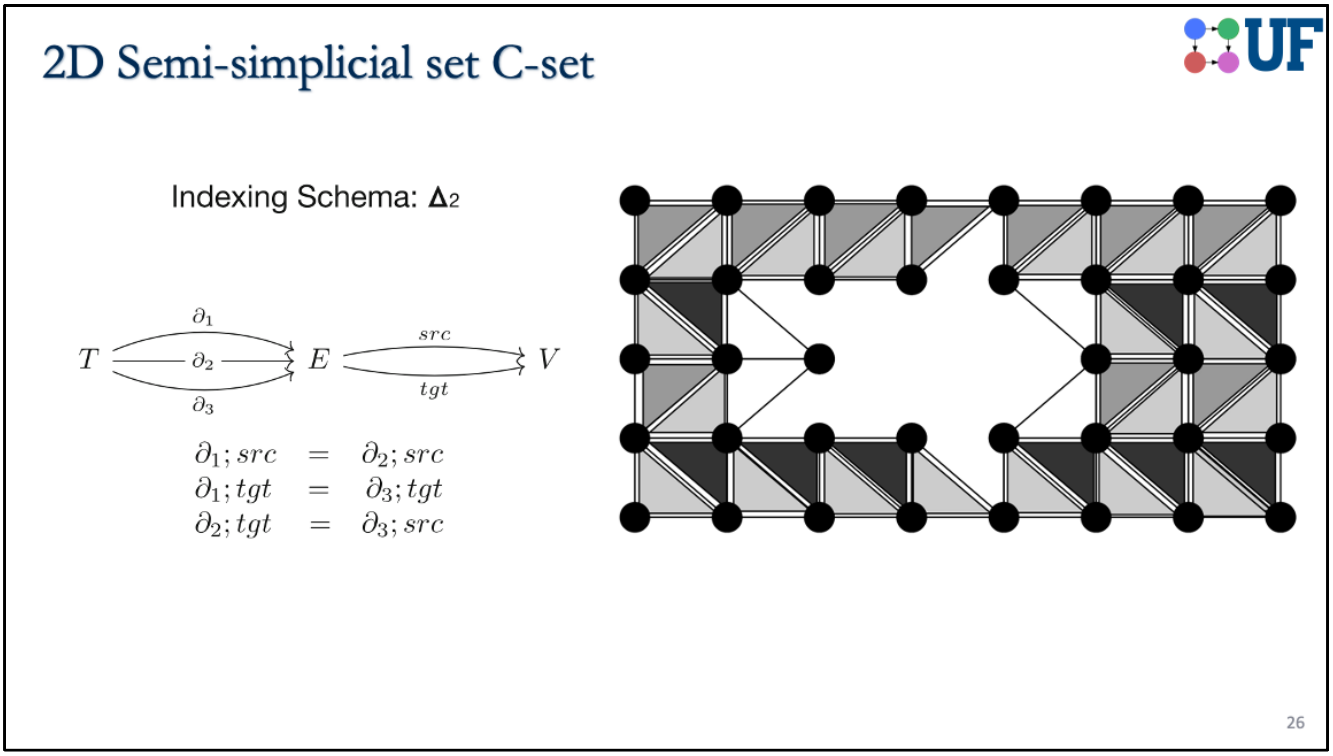
Bipartite multi digraphs

25

As a side note, despite the variety of things that can be expressed, Julia allows for these data types to be implemented efficiently.

For example, consider the C-set which encodes sparse graphs

This was benchmarked against Julia's native graph library, Lightgraphs, and performed competitively.



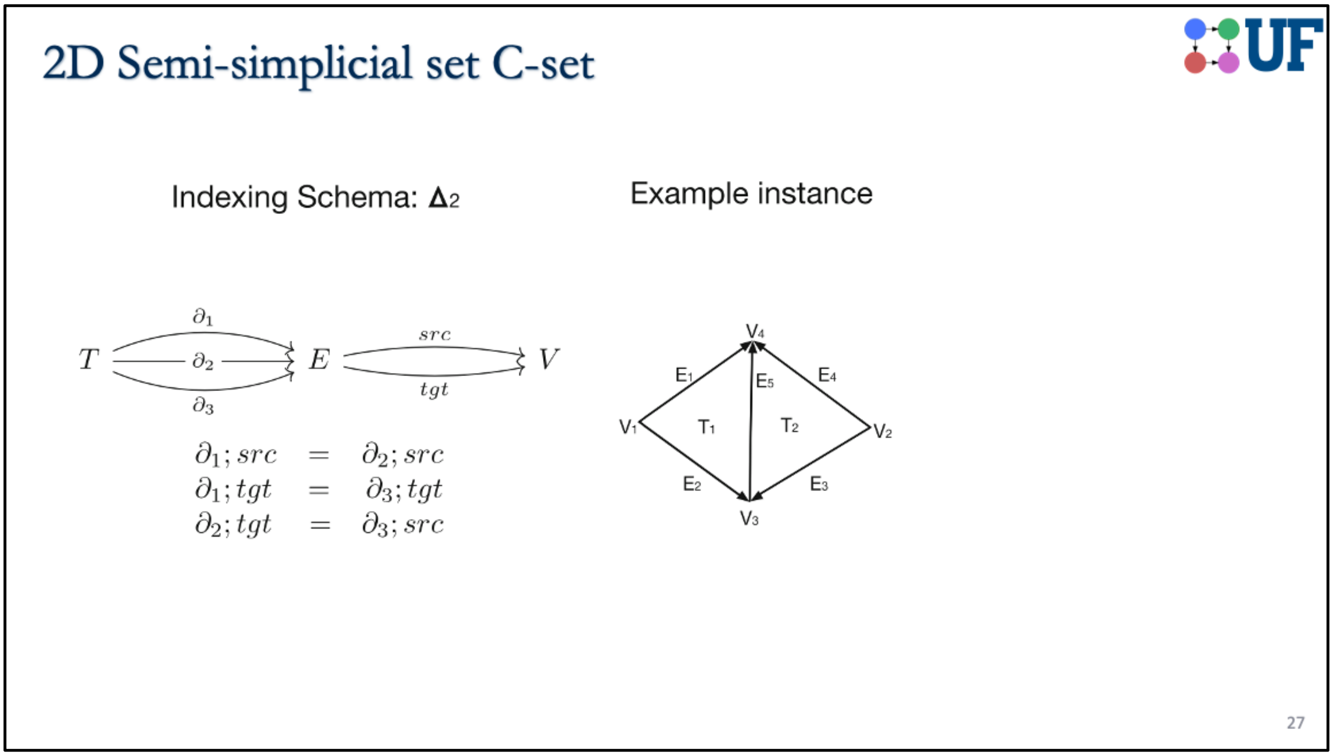
2D Semi-simplicial set C-set

Indexing Schema: **Δ**2

$$\begin{tikzcd} T \arrow[r, bend left, "\partial_1"] \arrow[r, "\partial_2"] \arrow[r, bend right, "\partial_3"'] & E \arrow[r, bend left, "src"] \arrow[r, bend right, "tgt"'] & V \end{tikzcd}$$

$$\begin{aligned} \partial_1; src &= \partial_2; src \\ \partial_1; tgt &= \partial_3; tgt \\ \partial_2; tgt &= \partial_3; src \end{aligned}$$

Just two more examples. This first one would be a C-set to define two-dimensional semisimplicial sets. This allows us to talk about graphs where SOME triples of edges can form triangles.



2D Semi-simplicial set C-set

Indexing Schema: Δ_2

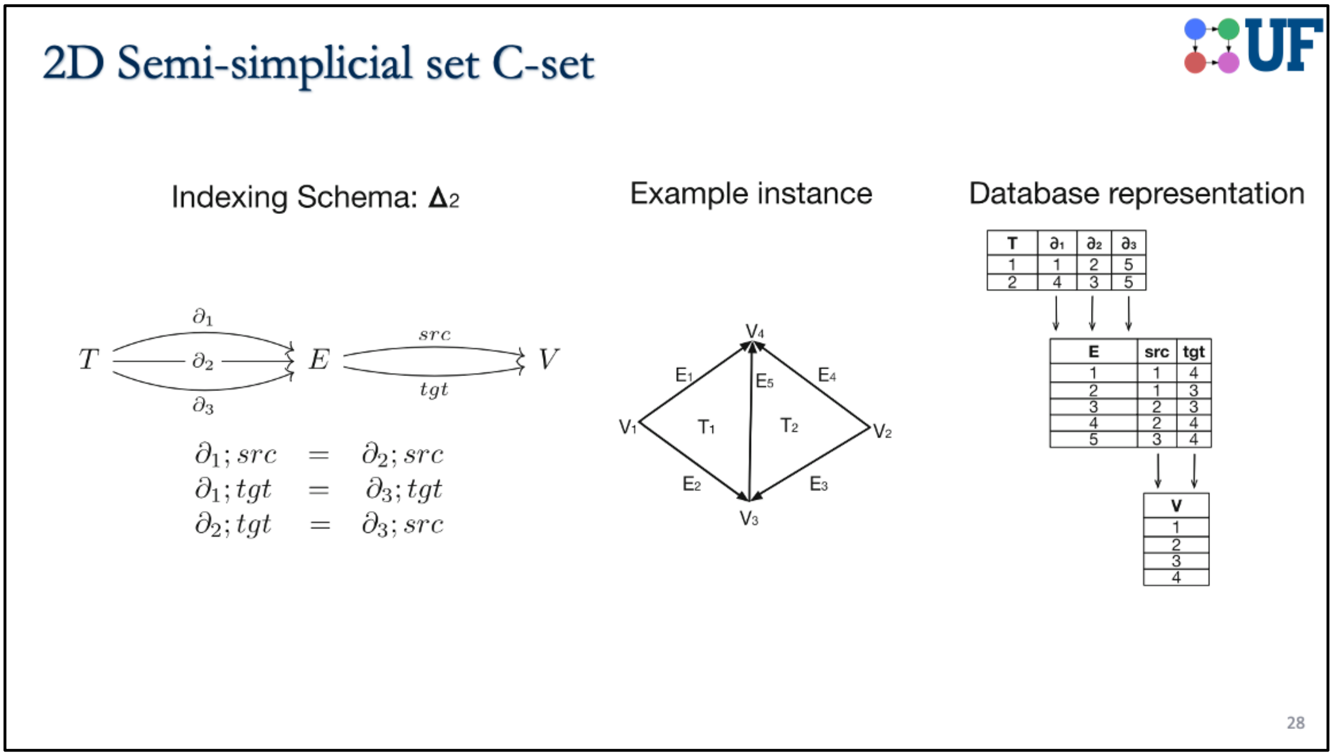
$$\begin{tikzcd}
T \arrow[r, bend left, "\partial_1"] \arrow[r, "\partial_2" description] \arrow[r, bend right, "\partial_3"'] & E \arrow[r, bend left=10, "src"] \arrow[r, bend right=10, "tgt"'] & V
\end{tikzcd}$$

$$\begin{aligned}
\partial_1; src &= \partial_2; src \\
\partial_1; tgt &= \partial_3; tgt \\
\partial_2; tgt &= \partial_3; src
\end{aligned}$$

Example instance

27

Here’s a smaller example of something we can represent. Two triangles which share one of their edges.



2D Semi-simplicial set C-set

Indexing Schema: Δ_2

$$\begin{tikzcd}
T \arrow[r, bend left, "\partial_1"] \arrow[r, "\partial_2" description] \arrow[r, bend right, "\partial_3"'] & E \arrow[r, bend left, "src"] \arrow[r, bend right, "tgt"'] & V
\end{tikzcd}$$

$$\begin{aligned}
\partial_1; src &= \partial_2; src \\
\partial_1; tgt &= \partial_3; tgt \\
\partial_2; tgt &= \partial_3; src
\end{aligned}$$

Example instance

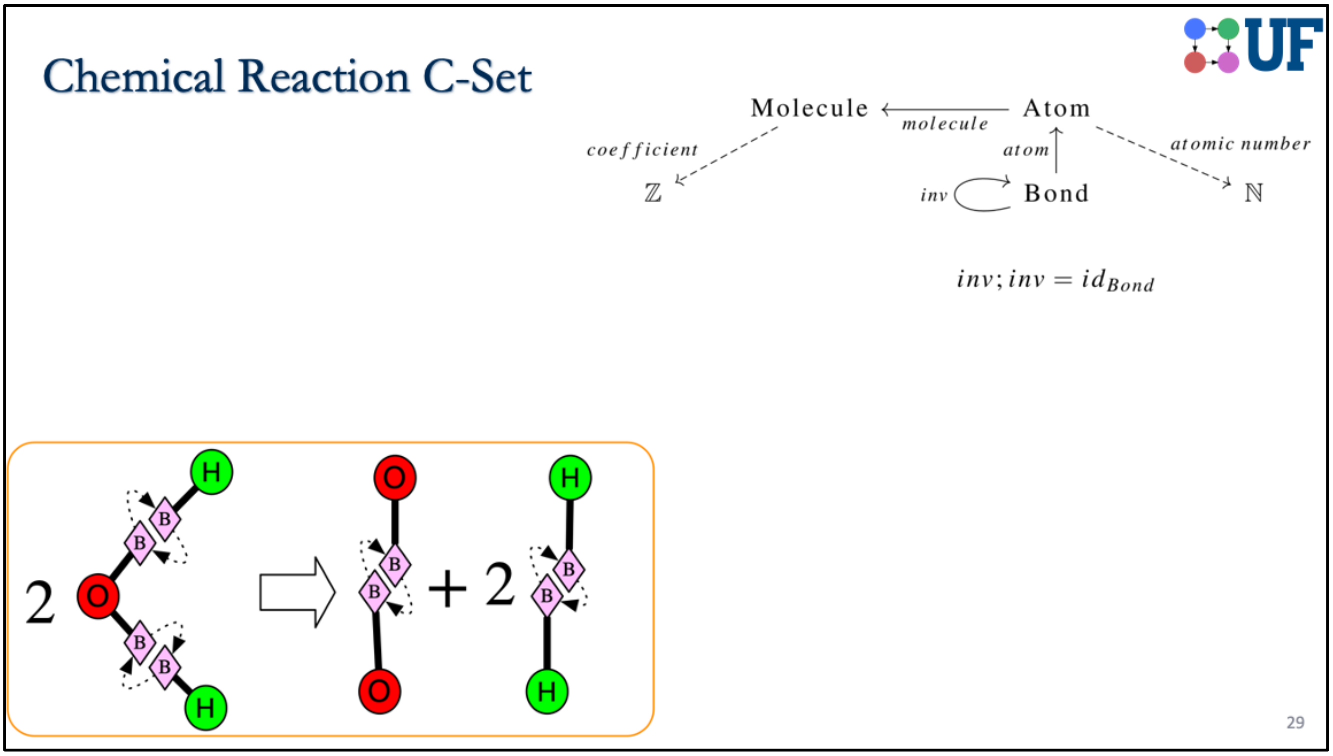
Database representation

T	∂1	∂2	∂3
1	1	2	5
2	4	3	5

E	src	tgt
1	1	4
2	1	3
3	2	3
4	2	4
5	3	4

V
1
2
3
4

The database style representation looks like this. Note we have two triangles, five edges, and four vertices.



Chemical Reaction C-Set

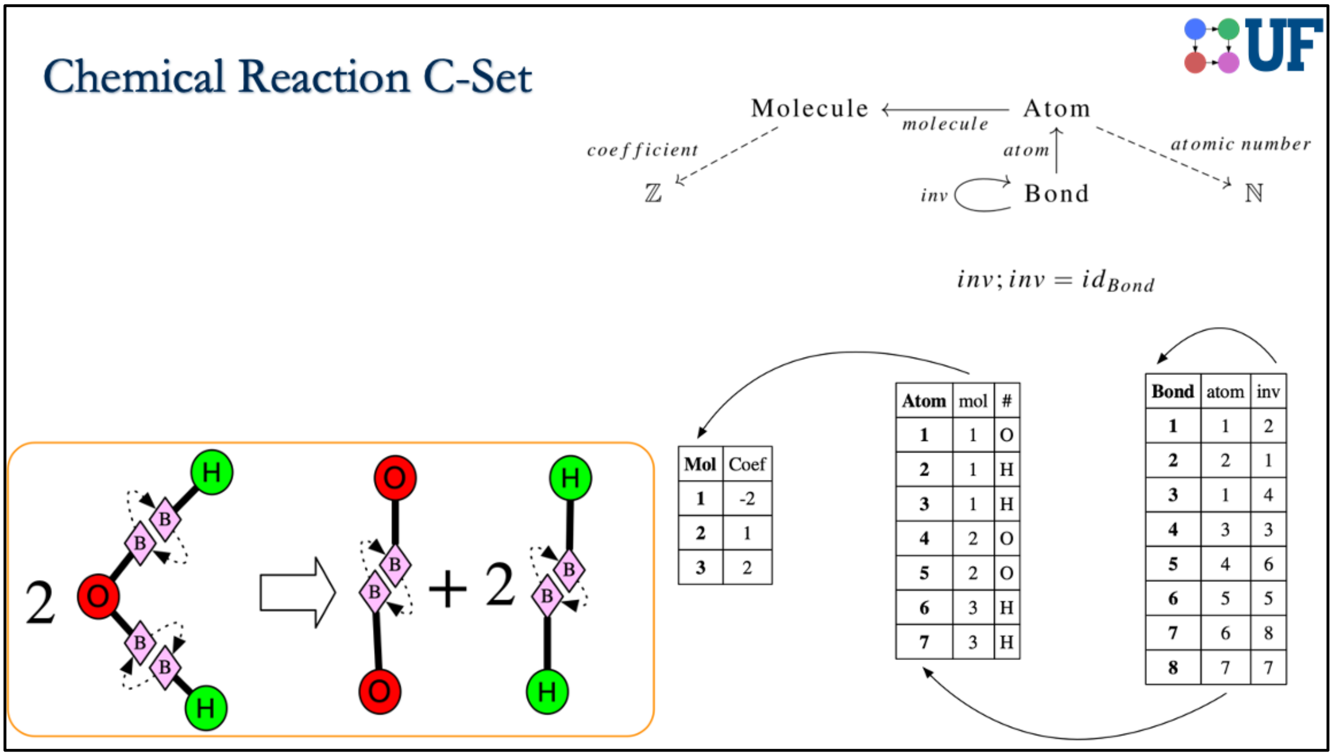
$$\begin{tikzcd}
& \text{Molecule} \arrow[dl, dashed, "coefficient"'] & \text{Atom} \arrow[l, "molecule"] \arrow[dr, dashed, "atomic\ number"] & \\
\mathbb{Z} & & \text{Bond} \arrow[u, "atom"] \arrow[loop left, "inv"] & \mathbb{N}
\end{tikzcd}$$

$$inv; inv = id_{Bond}$$

It’s important to build tools at the level of generality of C-sets rather than just graphs because now we can faithfully represent these chemical reactions.

For example, consider this schema. We’re saying there are many atoms per molecule and many bonds per atom. We can do a little trick to ensure that the bonds are symmetric by having two elements of the bond table correspond to each bond we visualize. We give a function “inv” which points these pairs of bonds at each other, using the equation inv;inv=id to enforce this property.

DRAW THE HALF EDGE BOND



Chemical Reaction C-Set

$$\begin{tikzcd}
& \text{Molecule} \arrow[dl, dashed, "coefficient"'] & \text{Atom} \arrow[l, "molecule"] \arrow[dr, dashed, "atomic\ number"] & \\
\mathbb{Z} & & \text{Bond} \arrow[u, "atom"'] \arrow[loop left, "inv"] & \mathbb{N}
\end{tikzcd}$$

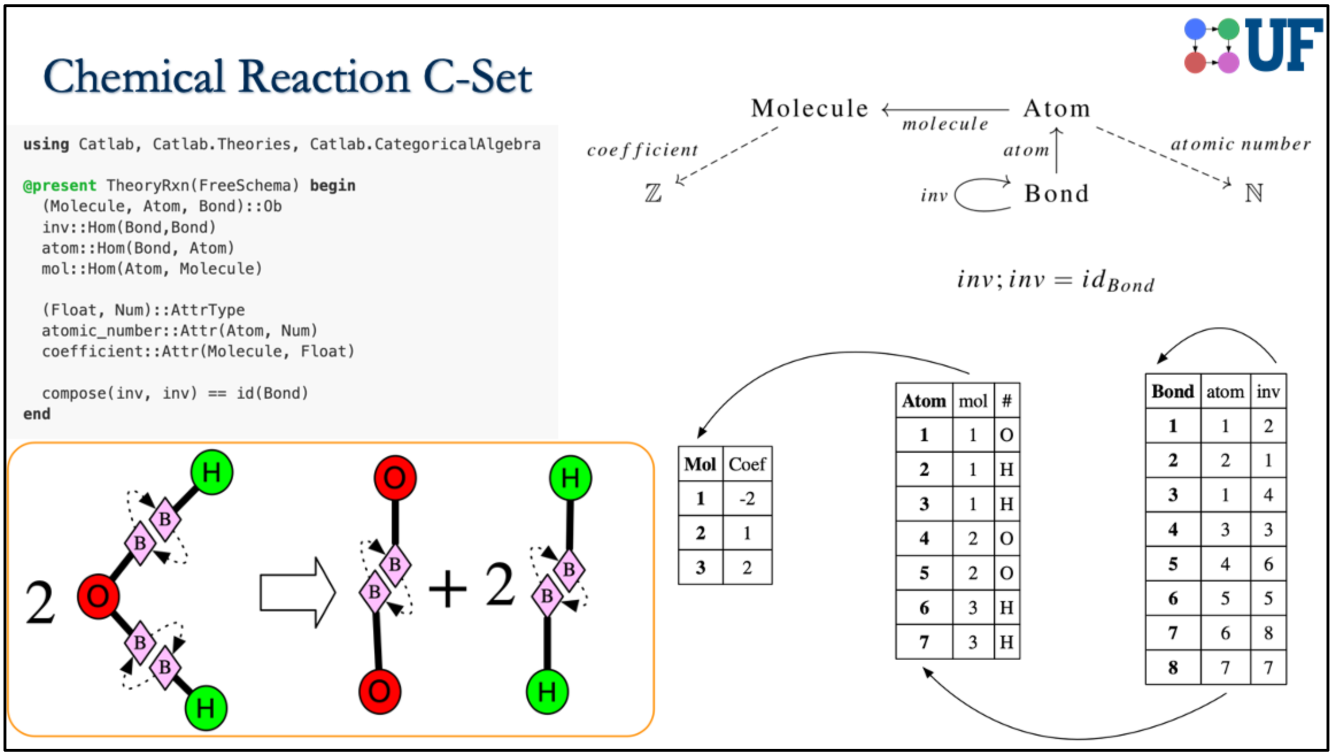
$$inv; inv = id_{Bond}$$

Mol	Coef
1	-2
2	1
3	2

Atom	mol	#
1	1	O
2	1	H
3	1	H
4	2	O
5	2	O
6	3	H
7	3	H

Bond	atom	inv
1	1	2
2	2	1
3	1	4
4	3	3
5	4	6
6	5	5
7	6	8
8	7	7

So to see it in action, here is the example we've been looking at viewed as a C-Set.



Chemical Reaction C-Set

```
using Catlab, Catlab.Theories, Catlab.CategoricalAlgebra

@present TheoryRxn(FreeSchema) begin
  (Molecule, Atom, Bond)::Ob
  inv::Hom(Bond,Bond)
  atom::Hom(Bond, Atom)
  mol::Hom(Atom, Molecule)

  (Float, Num)::AttrType
  atomic_number::Attr(Atom, Num)
  coefficient::Attr(Molecule, Float)

  compose(inv, inv) == id(Bond)
end
```

$$\begin{tikzcd}
 & \text{Molecule} \arrow[dl, dashed, "coefficient"'] & \text{Atom} \arrow[l, "molecule"'] \arrow[dr, dashed, "atomic\ number"] & \\
\mathbb{Z} & & \text{Bond} \arrow[u, "atom"] \arrow[loop left, "inv"] & \mathbb{N}
\end{tikzcd}$$

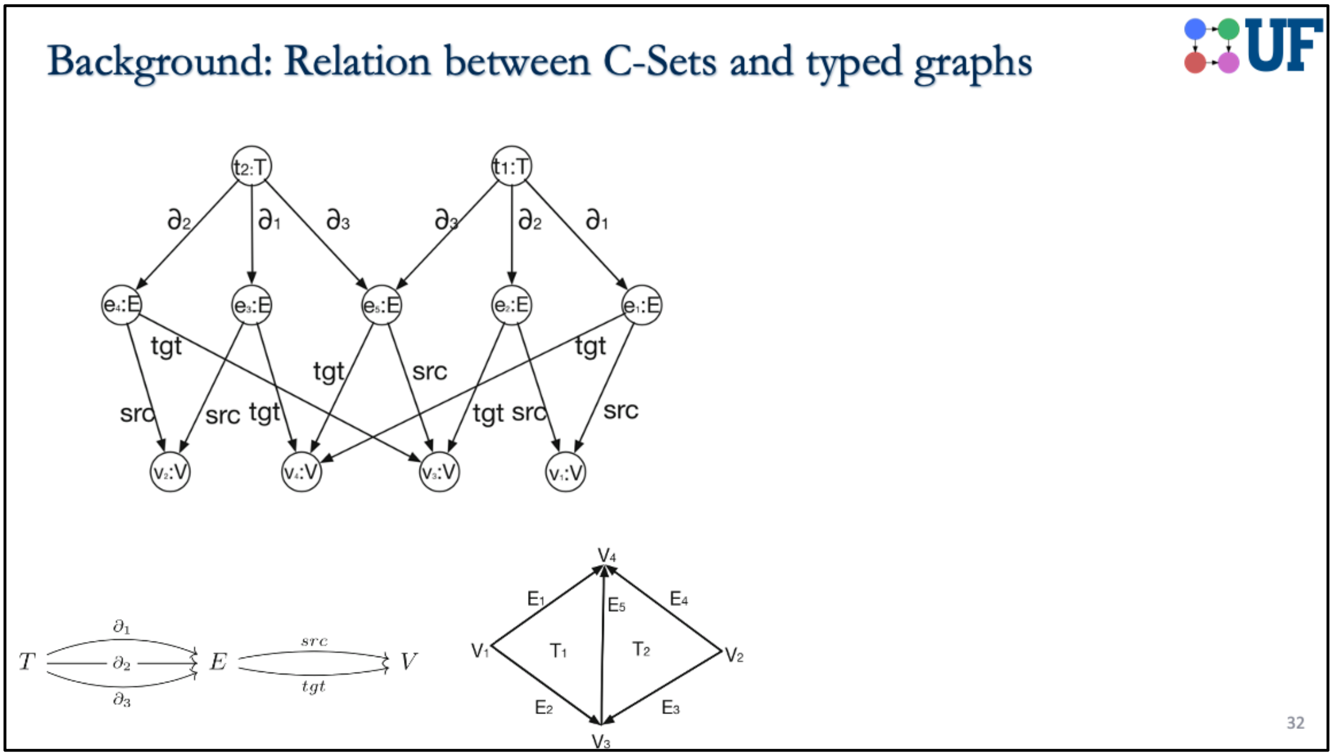
$$inv; inv = id_{Bond}$$

Mol	Coef
1	-2
2	1
3	2

Atom	mol	#
1	1	O
2	1	H
3	1	H
4	2	O
5	2	O
6	3	H
7	3	H

Bond	atom	inv
1	1	2
2	2	1
3	1	4
4	3	3
5	4	6
6	5	5
7	6	8
8	7	7

Also, we can look at the Catlab code for declaring this schema. This software allows us to declare C-sets and perform category theoretic constructions using them.

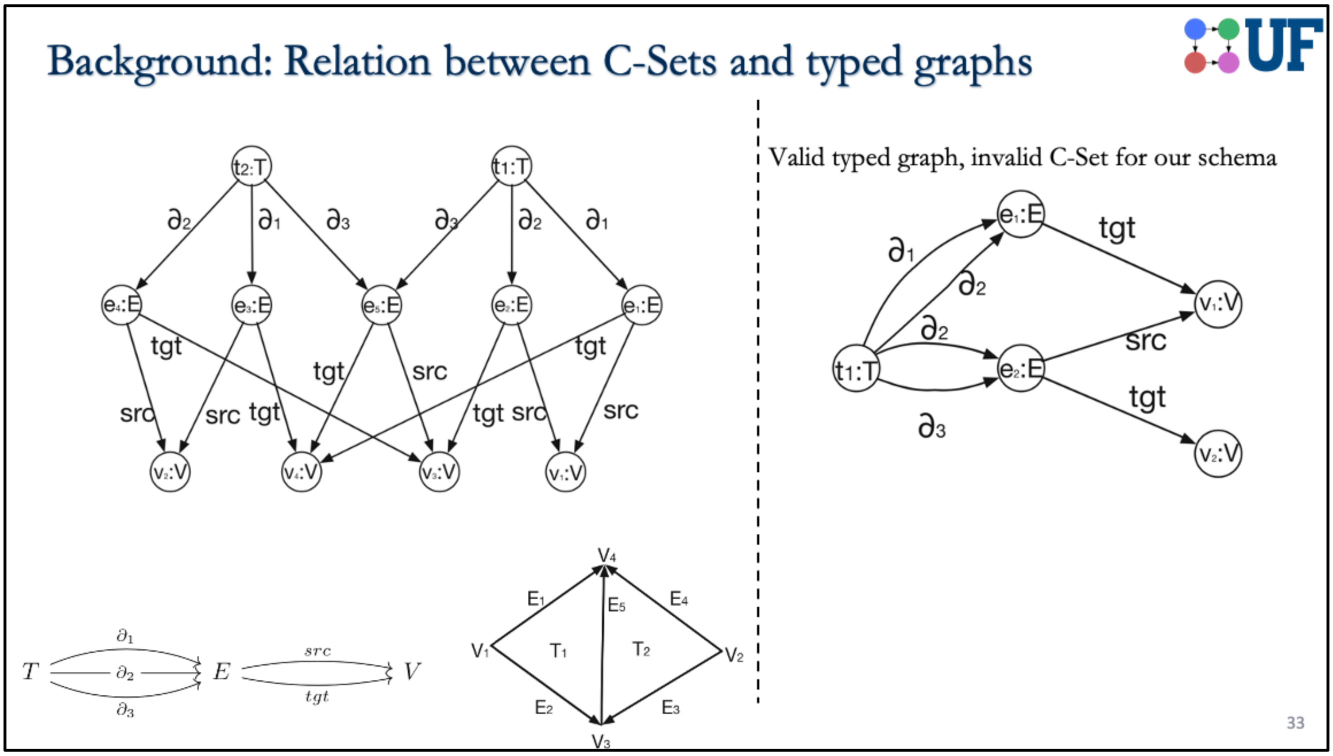


Background: Relation between C-Sets and typed graphs

$$\begin{tikzcd}
T \arrow[r, bend left, "\partial_1"] \arrow[r, "\partial_2"] \arrow[r, bend right, "\partial_3"'] & E \arrow[r, bend left, "src"] \arrow[r, bend right, "tgt"'] & V
\end{tikzcd}$$

C-sets are a generalization of typed graphs, although they are closely related. We can faithfully convert a C-Set like the one on the bottom into the typed graph above. Because this conversion is faithful in a way that category theory can make precise, many algorithms that operate on typed graphs will compute the correct thing for C-sets.

On the right, there is a valid typed graph but it’s not a valid C-set for a variety of reasons, for example there are multiple edges assigned as delta-2 of t1, and e1 has zero vertices assigned as its source.

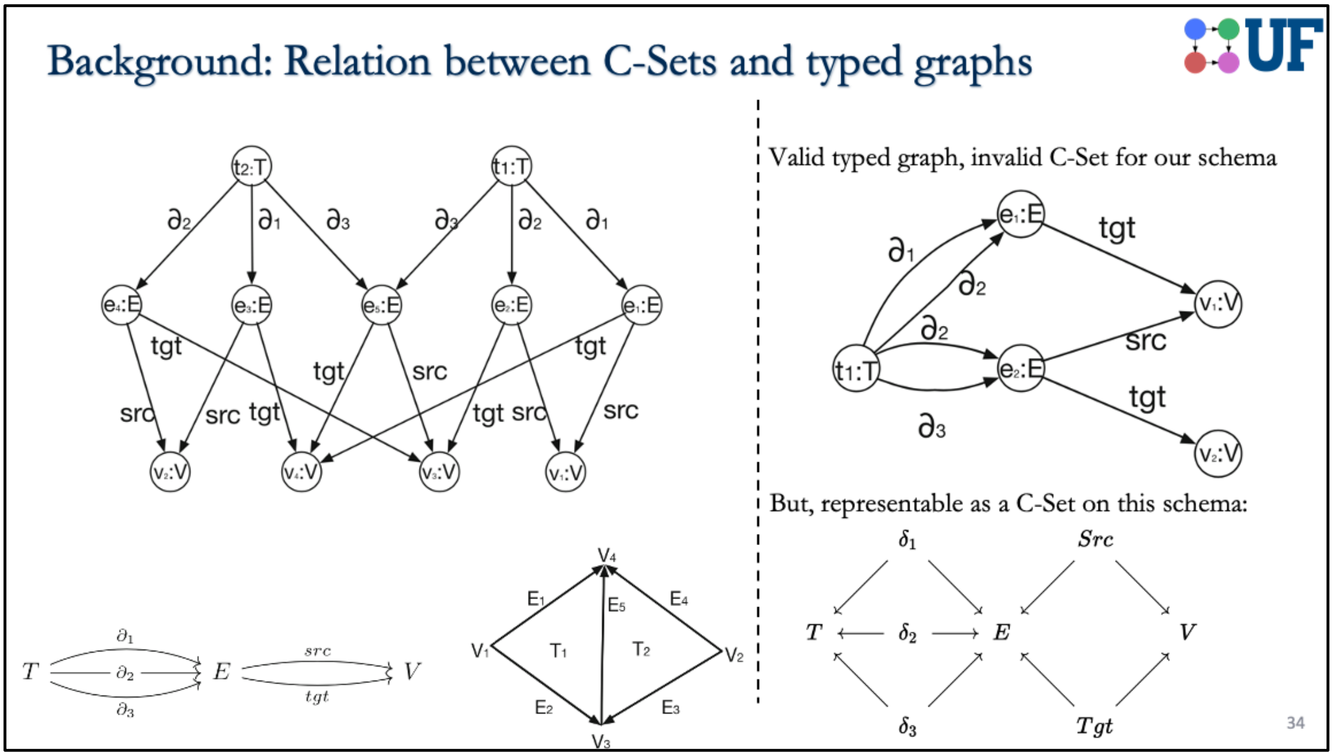


Background: Relation between C-Sets and typed graphs

$$\begin{tikzcd} T \arrow[r, bend left, "\partial_1"] \arrow[r, "\partial_2"] \arrow[r, bend right, "\partial_3"'] & E \arrow[r, bend left, "src"] \arrow[r, bend right, "tgt"'] & V \end{tikzcd}$$

Valid typed graph, invalid C-Set for our schema

On the right, there is a valid typed graph but it’s not a valid C-set for a variety of reasons, for example there are multiple edges assigned as delta-2 of t1, and e1 has zero vertices assigned as its source.



Background: Relation between C-Sets and typed graphs

$$\begin{tikzcd}
T \arrow[r, bend left, "\partial_1"] \arrow[r, "\partial_2"] \arrow[r, bend right, "\partial_3"'] & E \arrow[r, bend left, "src"] \arrow[r, bend right, "tgt"'] & V
\end{tikzcd}$$

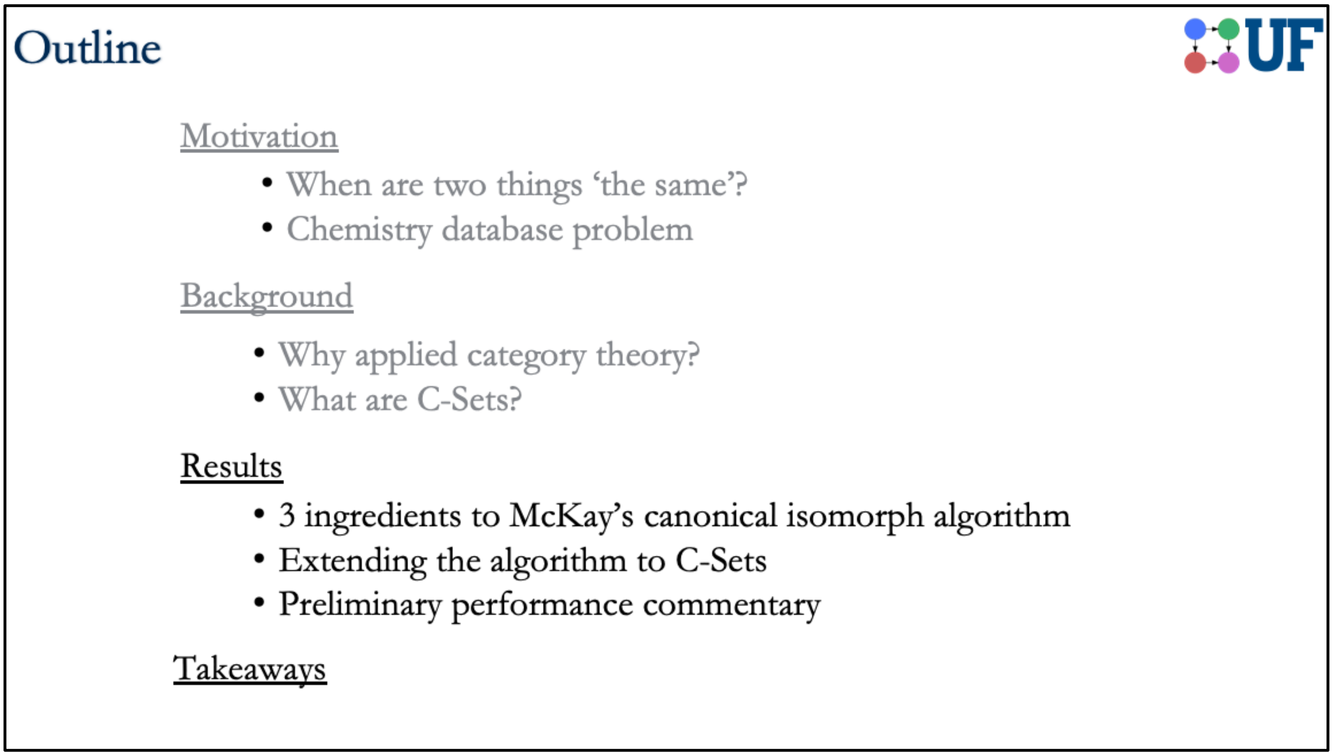
Valid typed graph, invalid C-Set for our schema

But, representable as a C-Set on this schema:

$$\begin{tikzcd}
 & \delta_1 \arrow[dl] \arrow[dr] & & Src \arrow[dl] \arrow[dr] & \\
T & \delta_2 \arrow[l] \arrow[r] & E & & V \\
 & \delta_3 \arrow[ul] \arrow[ur] & & Tgt \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

34

It is a valid C-set on this schema, where we've replaced all the arrows with a span of arrows, i.e. replaced functions with relations. So while C-sets can handle the looseness of relations by choice, it is more expressive because it can encode the constraint of functions.



Outline

Motivation

- When are two things 'the same'?
- Chemistry database problem

Background

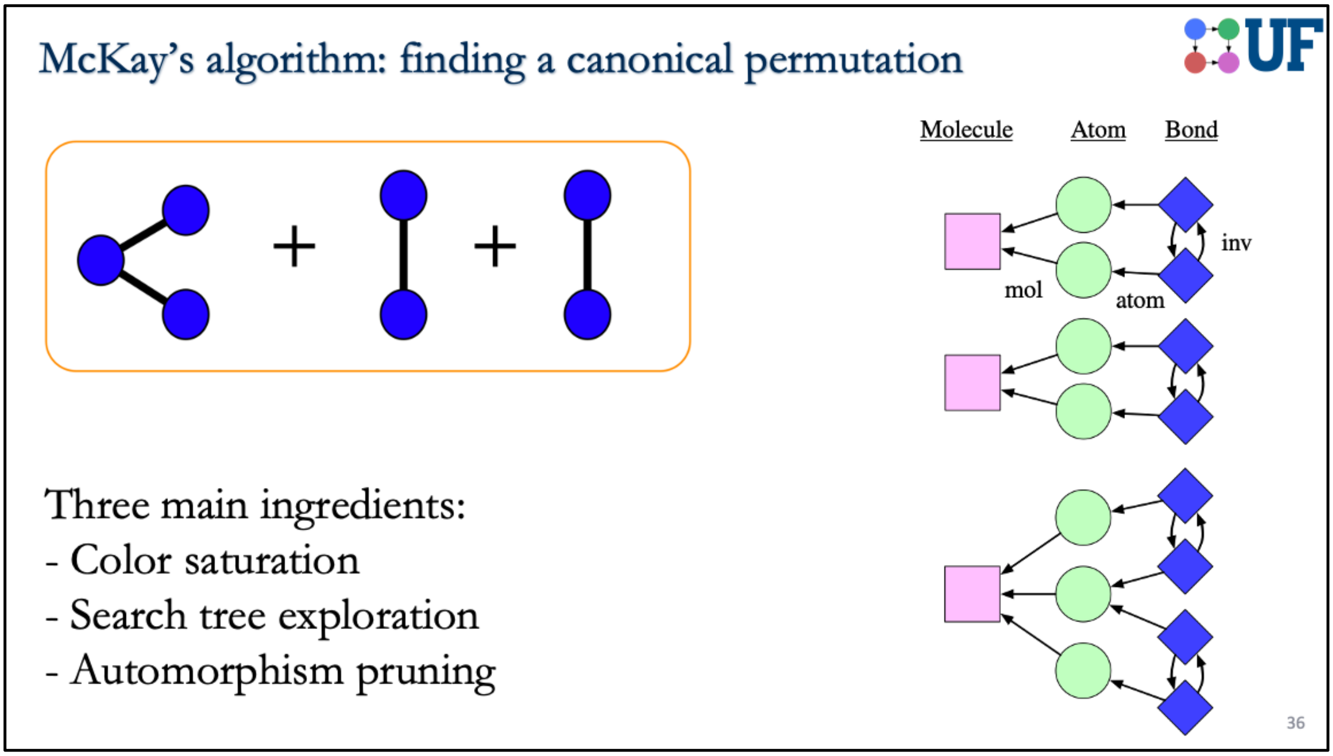
- Why applied category theory?
- What are C-Sets?

Results

- 3 ingredients to McKay's canonical isomorph algorithm
- Extending the algorithm to C-Sets
- Preliminary performance commentary

Takeaways

13 min



McKay's algorithm: finding a canonical permutation

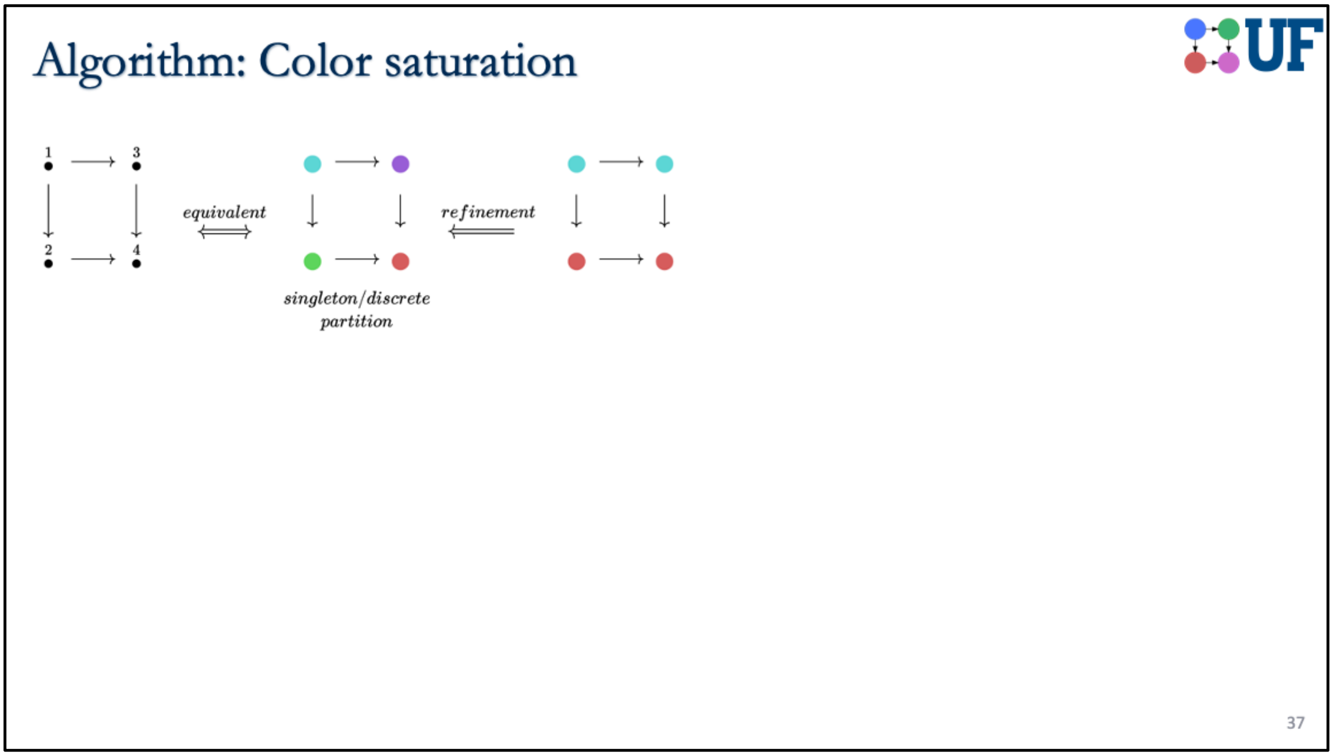
+
+
Molecule
Atom
Bond
inv
mol
atom

Three main ingredients:
- Color saturation
- Search tree exploration
- Automorphism pruning

We'll focus on the case where there are no attributes (which are very easy to incorporate).

So this algorithm computes a canonical ordering of vertices in the case of a simple graph.

For a C-Set, a automorphism translates into a separate permutation of each of the underlying sets that preserves the structure.



Algorithm: Color saturation

Automorphism is a bijection (every vertex has own color) – here we have surjections – mult have same color

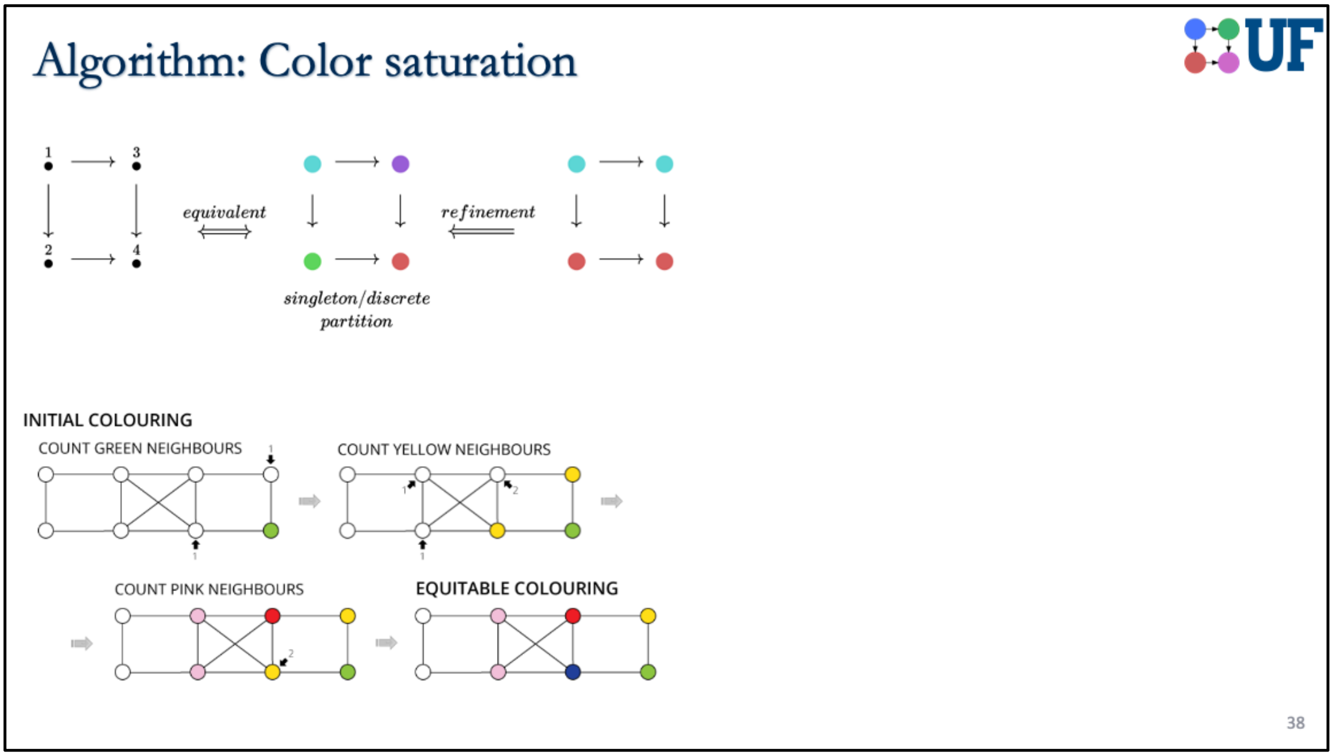
Colorings of graphs are a kind of generalization of permutations. If our colors are ordered, then a permutation is when every vertex has a different color, but maybe we don't have a complete permutation yet, we just have partial information. So that Here if I know the top two vertices come before the bottom two but I don't know the relative ordering within each group, I could characterize this by coloring the two pairs like so.

As I learn more information, I can refine this partitioning and get a fine coloring, which eventually will lead to a partition.

So color saturation is a procedure which takes in a coloring attempts to refine it, i.e. bring us closer to a permutation. It does this by looking at the local connectivity info, for example, here the corner is green, and I can then use this to distinguish the edges which have zero green neighbors or one green neighbor. This allows me then distinguish things again, and so on until this process hits a dead end, which is called an equitable coloring.

So that's how it works for graphs. C-sets also have a notion of local connectivty,

which is apparent when you view a C-set as a kind of typed graph. We can consider the local connectivity info which is used to distinguish vertices as its in neighbors and its out neighbors for each kind of edge.



Algorithm: Color saturation

INITIAL COLOURING

COUNT GREEN NEIGHBOURS

COUNT YELLOW NEIGHBOURS

COUNT PINK NEIGHBOURS

EQUITABLE COLOURING

38

Automorphism is a bijection (every vertex has own color) – here we have surjections – mult have same color

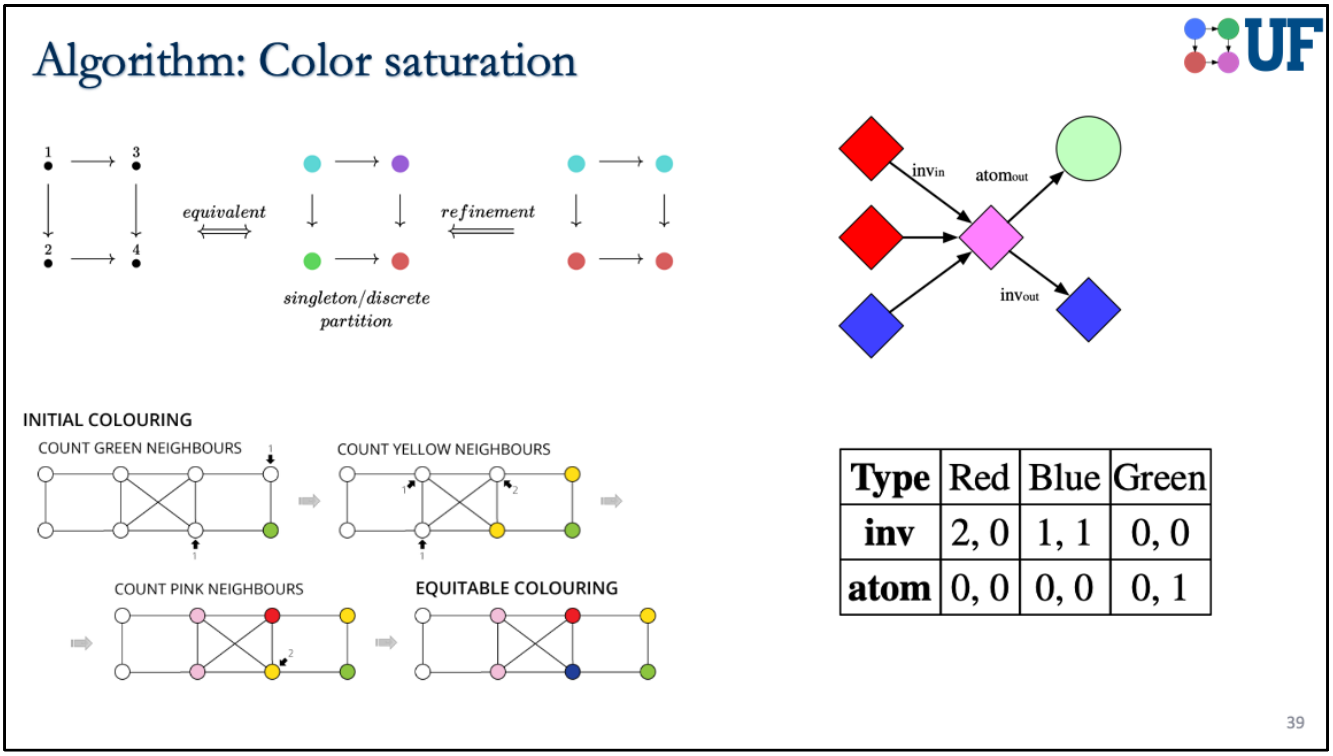
Colorings of graphs are a kind of generalization of permutations. If our colors are ordered, then a permutation is when every vertex has a different color, but maybe we don't have a complete permutation yet, we just have partial information. So that Here if I know the top two vertices come before the bottom two but I don't know the relative ordering within each group, I could characterize this by coloring the two pairs like so.

As I learn more information, I can refine this partitioning and get a fine coloring, which eventually will lead to a partition.

So color saturation is a procedure which takes in a coloring attempts to refine it, i.e. bring us closer to a permutation. It does this by looking at the local connectivity info, for example, here the corner is green, and I can then use this to distinguish the edges which have zero green neighbors or one green neighbor. This allows me then distinguish things again, and so on until this process hits a dead end, which is called an equitable coloring.

So that's how it works for graphs. C-sets also have a notion of local connectivty,

which is apparent when you view a C-set as a kind of typed graph. We can consider the local connectivity info which is used to distinguish vertices as its in neighbors and its out neighbors for each kind of edge.



Algorithm: Color saturation

Type	Red	Blue	Green
inv	2, 0	1, 1	0, 0
atom	0, 0	0, 0	0, 1

Automorphism is a bijection (every vertex has own color) – here we have surjections – mult have same color

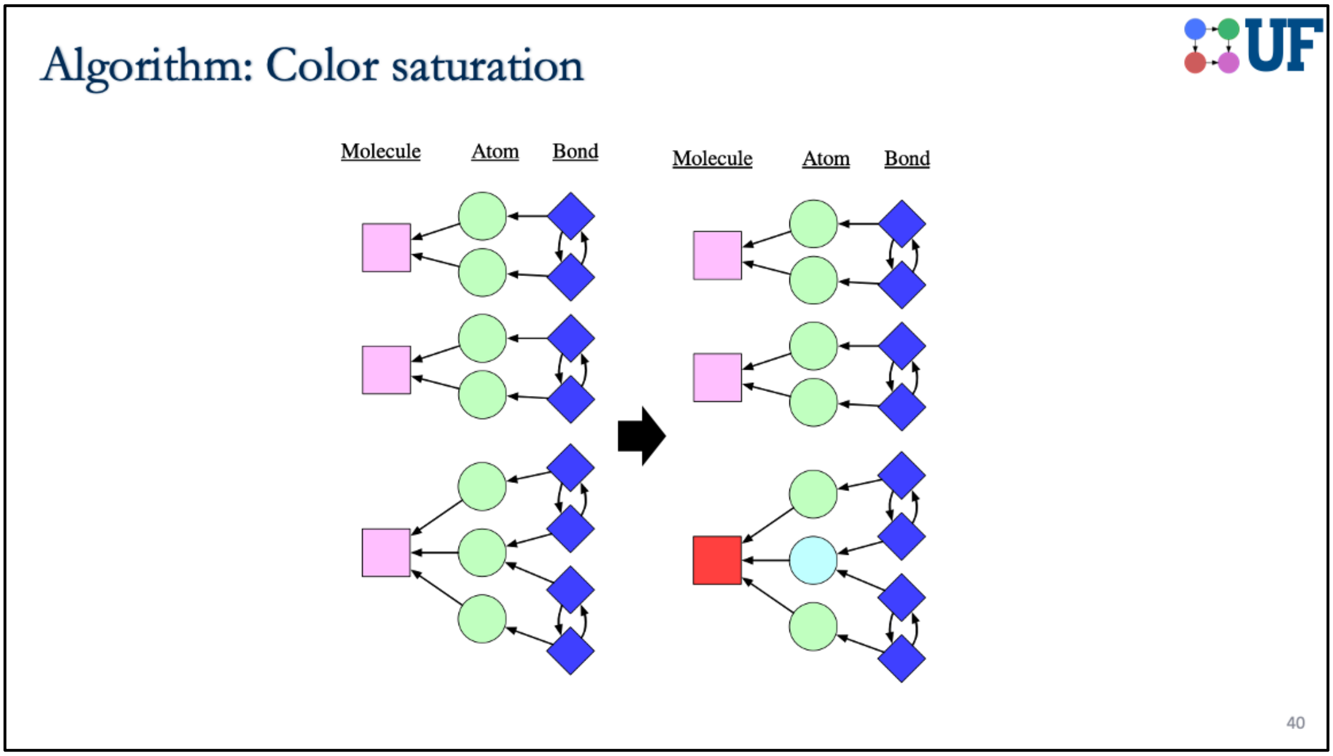
Colorings of graphs are a kind of generalization of permutations. If our colors are ordered, then a permutation is when every vertex has a different color, but maybe we don't have a complete permutation yet, we just have partial information. So that Here if I know the top two vertices come before the bottom two but I don't know the relative ordering within each group, I could characterize this by coloring the two pairs like so.

As I learn more information, I can refine this partitioning and get a fine coloring, which eventually will lead to a partition.

So color saturation is a procedure which takes in a coloring attempts to refine it, i.e. bring us closer to a permutation. It does this by looking at the local connectivity info, for example, here the corner is green, and I can then use this to distinguish the edges which have zero green neighbors or one green neighbor. This allows me then distinguish things again, and so on until this process hits a dead end, which is called an equitable coloring.

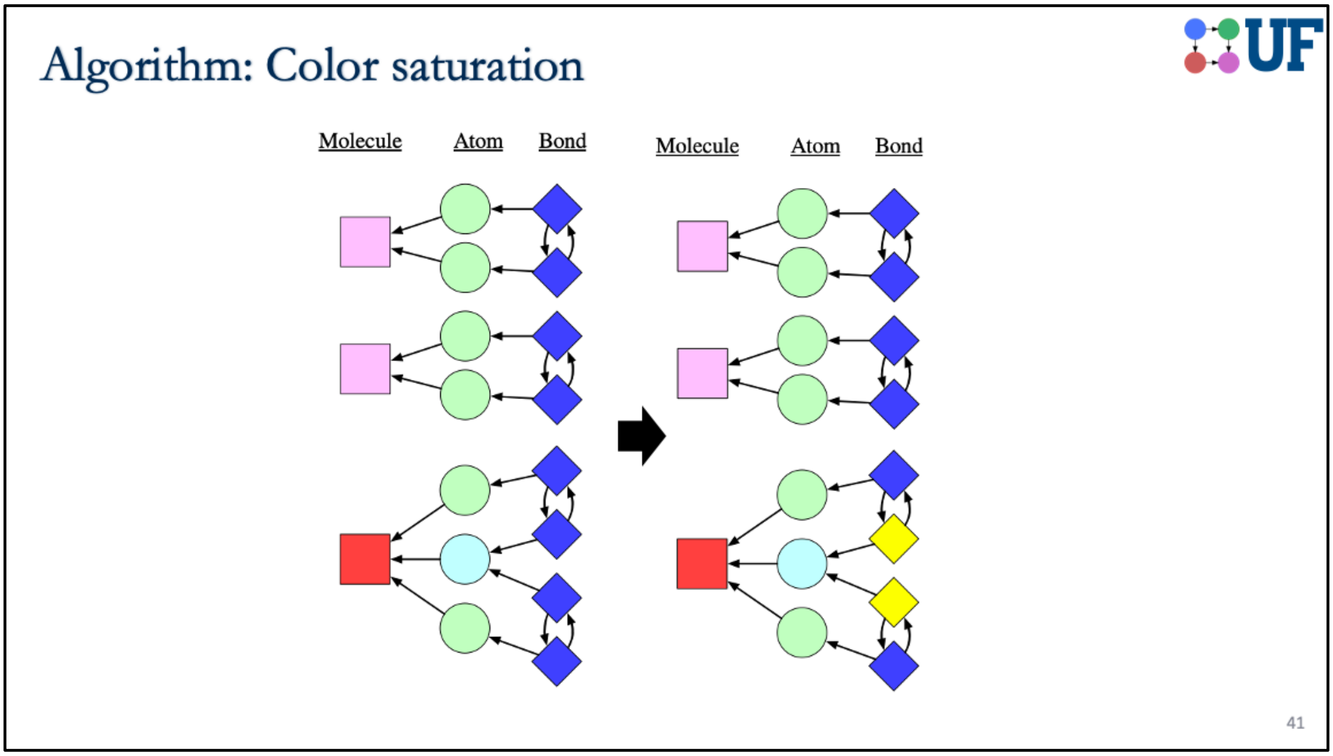
So that's how it works for graphs. C-sets also have a notion of local connectivty,

which is apparent when you view a C-set as a kind of typed graph. We can consider the local connectivity info which is used to distinguish vertices as its in neighbors and its out neighbors for each kind of edge.



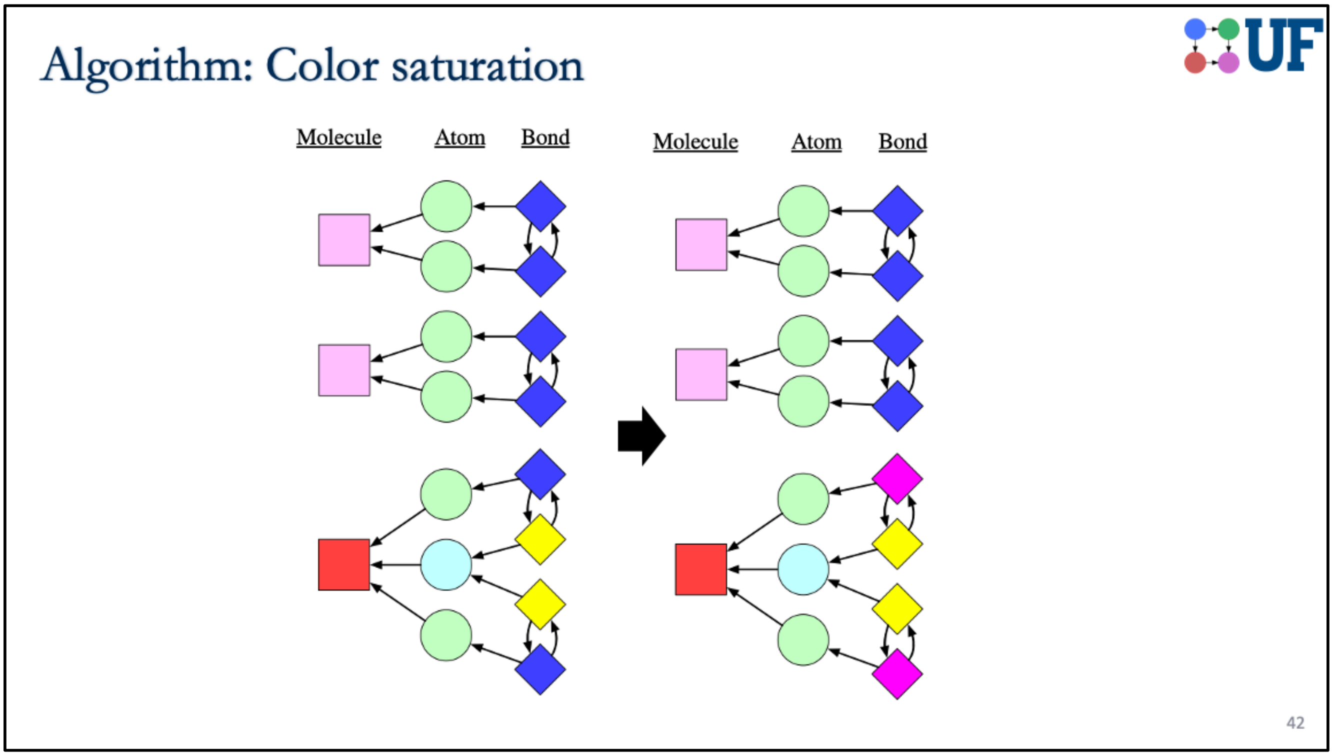
Algorithm: Color saturation

This is how color saturation would work with our initial data, which has perfect symmetry among each of the molecules, bonds, and atoms.



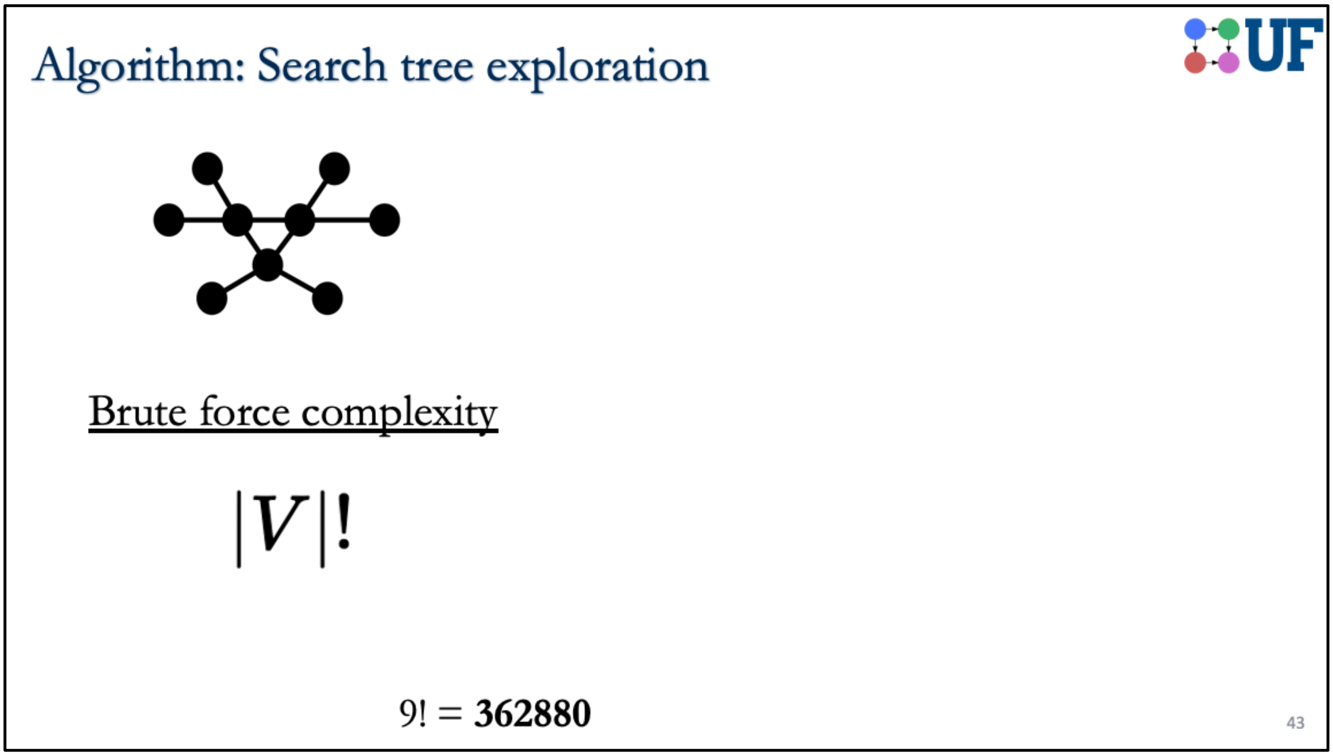
Algorithm: Color saturation

This is how color saturation would work with our initial data, which has perfect symmetry among each of the molecules, bonds, and atoms.



Algorithm: Color saturation

This is how color saturation would work with our initial data, which has perfect symmetry among each of the molecules, bonds, and atoms.



Algorithm: Search tree exploration

Brute force complexity

$$|V|!$$

$9! = \mathbf{362880}$

Search tree exploration is the next step of the algorithm.

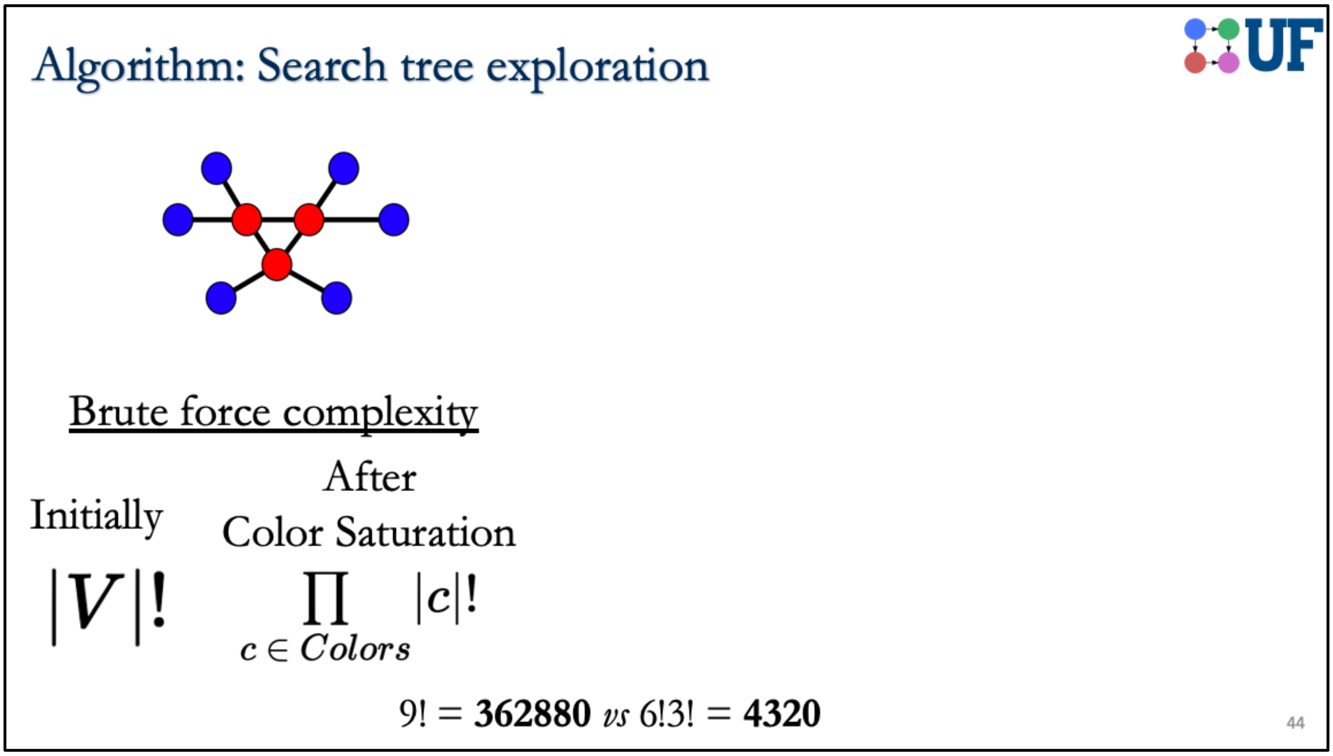
Here’s an example of a graph for which color saturation has hit a dead end. There is no way to use the structure of the graph to break the symmetry for us.

Our original problem was finding the automorphisms of a graph with V vertices, which could be solved brute force by looking at all V factorial permutations.

Color saturation reduced this somewhat by eliminating some of the possible permutations. We really can consider the colors independently, giving this formula.

So we have to artificially break the symmetry.

This incremental exploration of this set of permutations would not save us anything if it weren’t for the fact that we can run color saturation at every step, which significantly reduces the number of permutations we need to consider.



Algorithm: Search tree exploration

Brute force complexity

Initially

$$|V|!$$

After Color Saturation

$$\prod_{c \in Colors} |c|!$$

$9! = \mathbf{362880}$ *vs* $6!3! = \mathbf{4320}$

44

Search tree exploration is the next step of the algorithm.

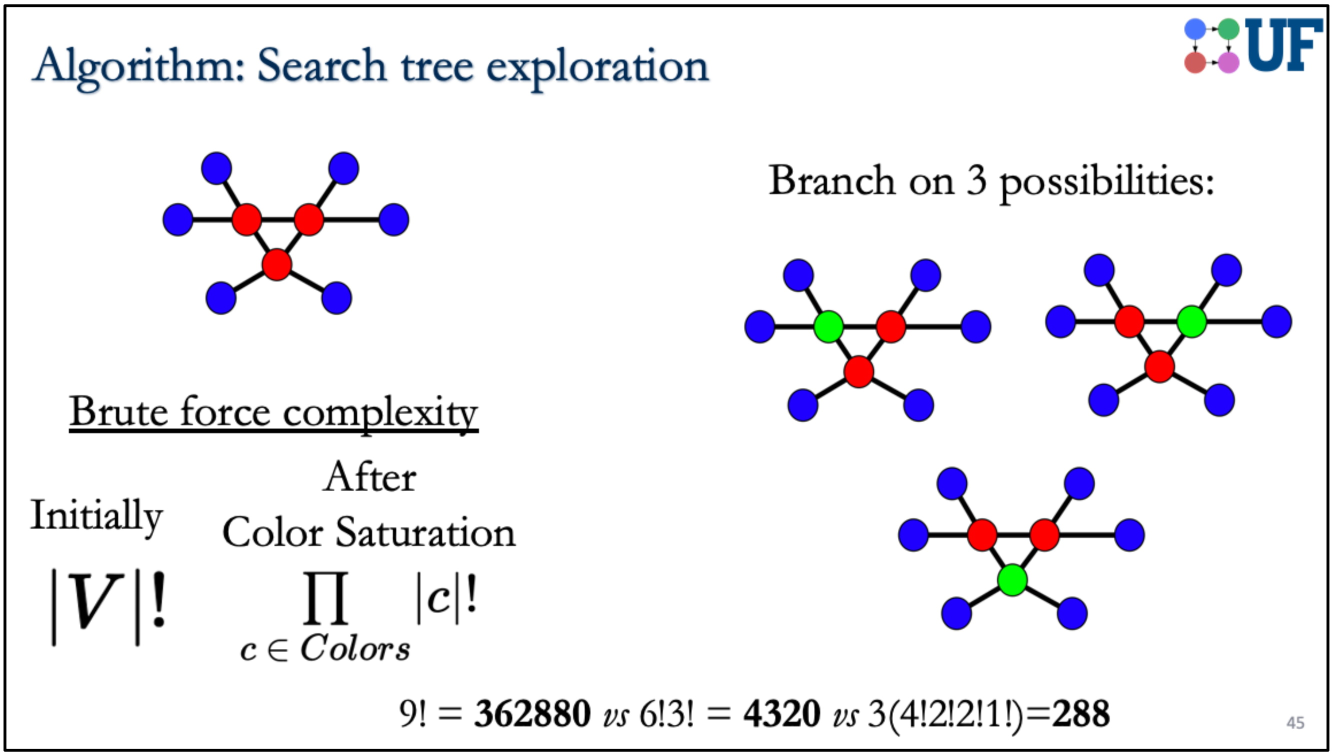
Here’s an example of a graph for which color saturation has hit a dead end. There is no way to use the structure of the graph to break the symmetry for us.

Our original problem was finding the automorphisms of a graph with V vertices, which could be solved brute force by looking at all V factorial permutations.

Color saturation reduced this somewhat by eliminating some of the possible permutations. We really can consider the colors independently, giving this formula.

So we have to artificially break the symmetry.

This incremental exploration of this set of permutations would not save us anything if it weren’t for the fact that we can run color saturation at every step, which significantly reduces the number of permutations we need to consider.



Algorithm: Search tree exploration

Branch on 3 possibilities:

Brute force complexity

Initially

$$|V|!$$

After Color Saturation

$$\prod_{c \in Colors} |c|!$$

9! = **362880** *vs* 6!3! = **4320** *vs* 3(4!2!2!1!)=**288**

45

Search tree exploration is the next step of the algorithm.

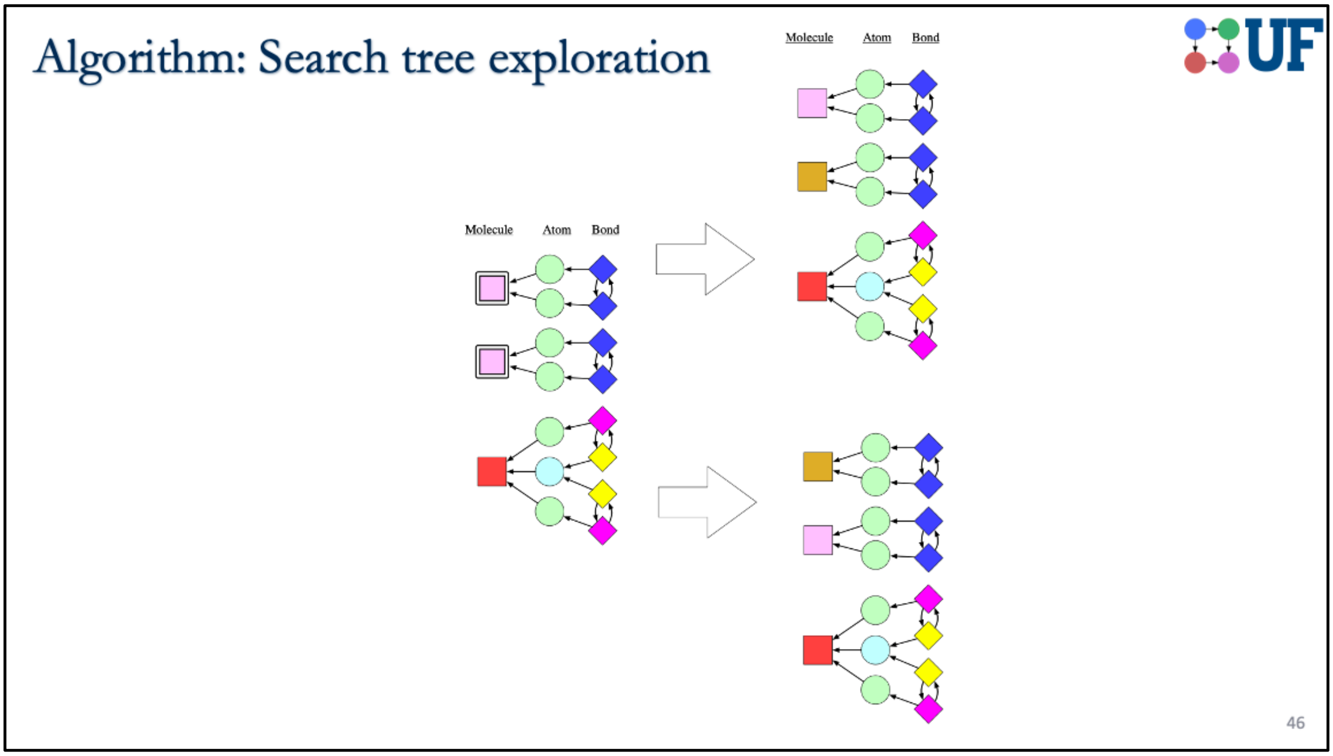
Here’s an example of a graph for which color saturation has hit a dead end. There is no way to use the structure of the graph to break the symmetry for us.

Our original problem was finding the automorphisms of a graph with V vertices, which could be solved brute force by looking at all V factorial permutations.

Color saturation reduced this somewhat by eliminating some of the possible permutations. We really can consider the colors independently, giving this formula.

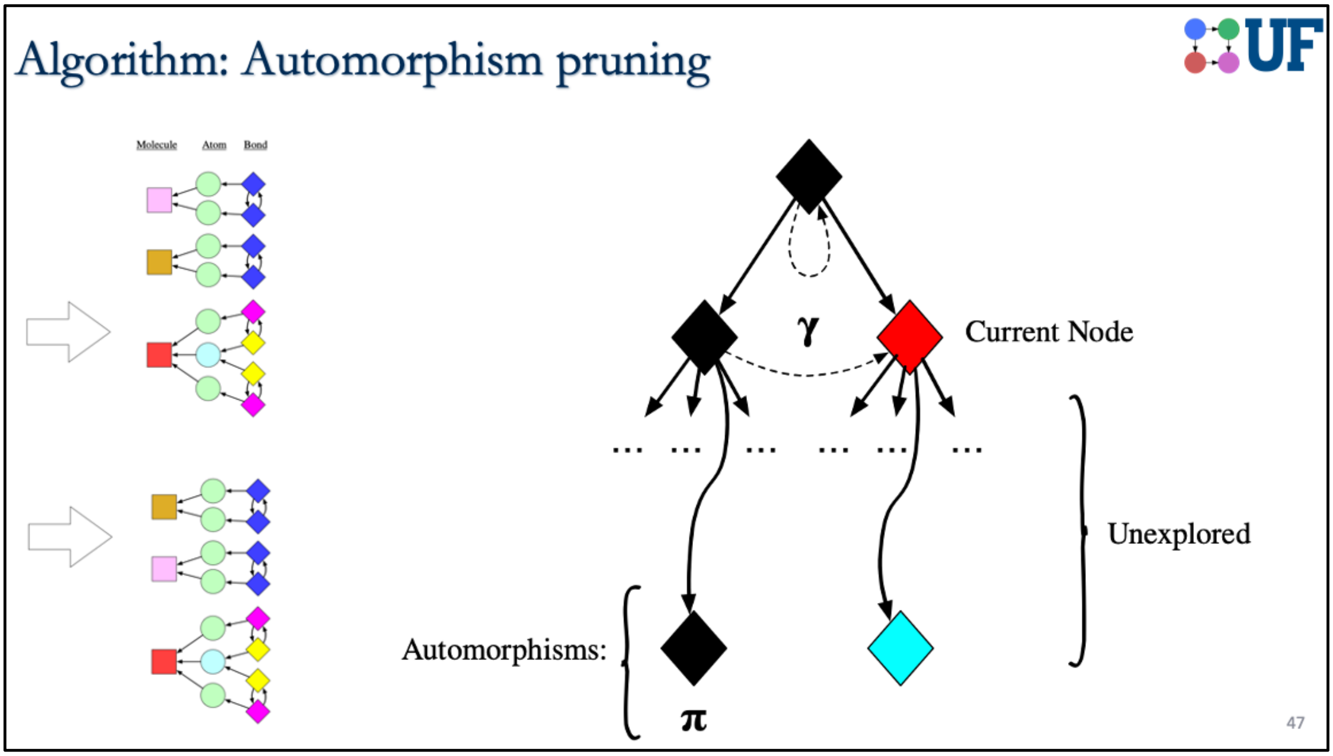
So we have to artificially break the symmetry.

This incremental exploration of this set of permutations would not save us anything if it weren’t for the fact that we can run color saturation at every step, which significantly reduces the number of permutations we need to consider.



Algorithm: Search tree exploration

So again it's a very straightforward generalization from graphs to C-sets because this idea of breaking a symmetry of a color extends nicely. We just have to break the symmetry of one of our sets (in this case, Molecules, Atoms, or Bonds) at a time.



Algorithm: Automorphism pruning

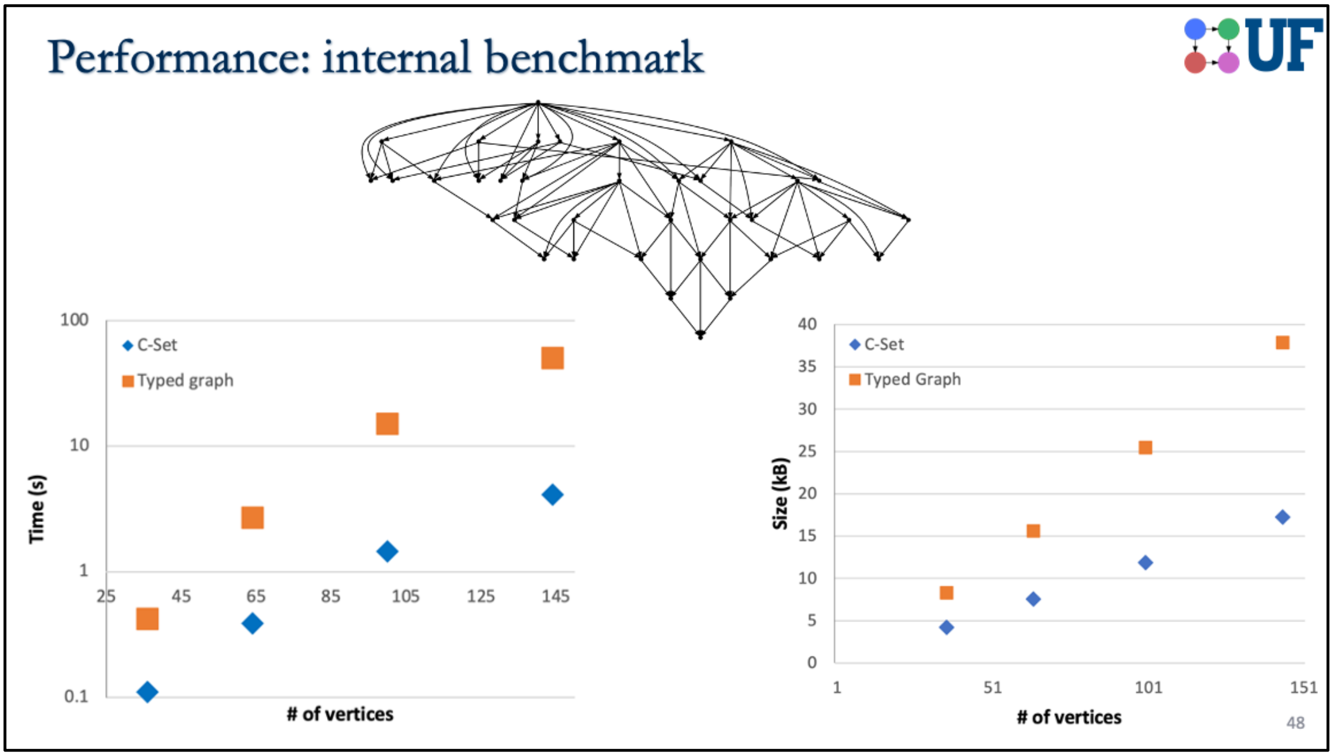
The last primary tactic in McKay's algorithm (though there are many other performance boosting techniques) is called automorphism pruning. This allows us to recognize a branch of the search tree as degenerate in some way, not really providing any new info.

On the previous slide, you might have noticed that we branched on two identical molecules. Intuitively, we are going to repeat work by treating those as completely independent. However, after exploring one of those two molecules depth first search style, we'll have found an automorphism which witnesses the fact that the two branches are equivalent, and essentially this notion of checking all the automorphisms you've found against a new branch you're about to explore will save you.

While this technique was developed with graph homomorphisms in mind, the argument works equally well for C-set homomorphisms, so we can use the technique, too.

Because the algorithm is algebraic (works with concepts like automorphism groups) – c-sets are the generalization of

Bipartite matching algorithms / augmenting paths – that part won’t be as easy to generalize.

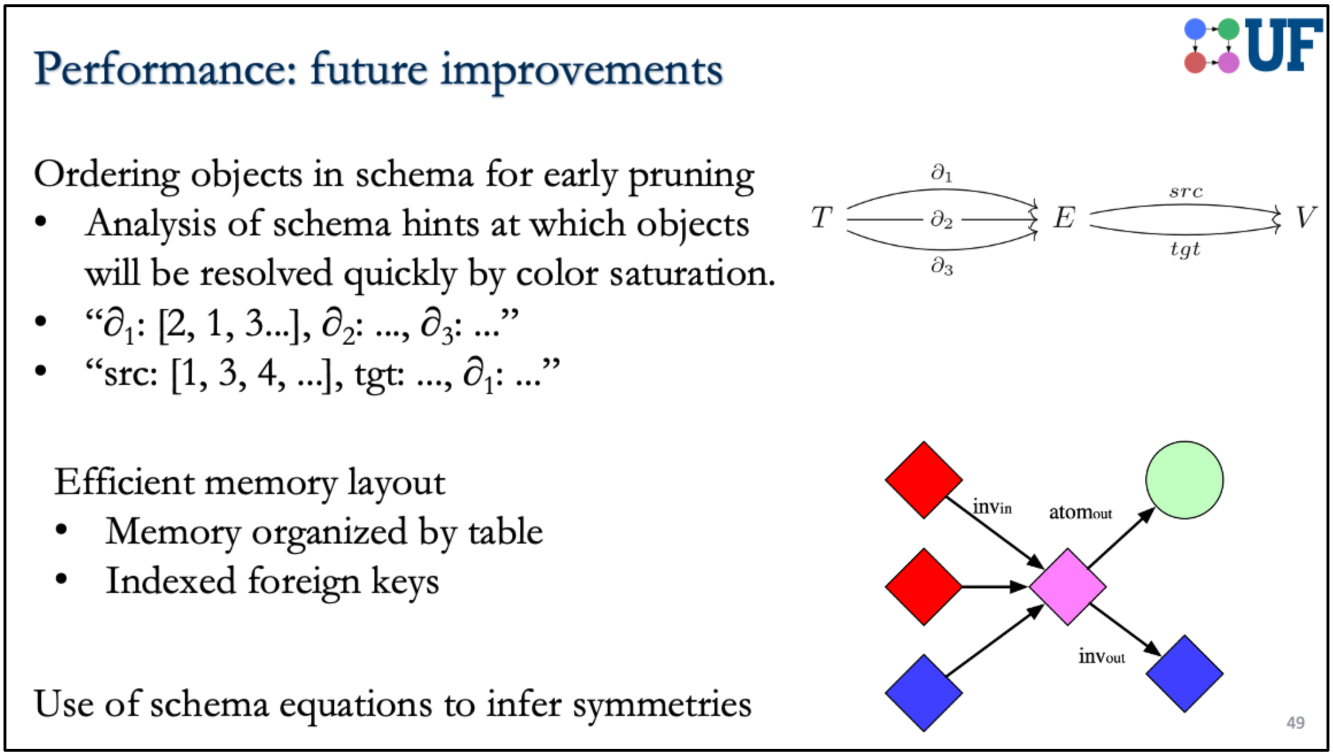


Performance: internal benchmark

Presently, a chemist with a rich data structure like their chemical reaction would have to convert their input into a graph and feed to nauty in order to use existing software.

So a simple benchmark we can perform is to compare how our algorithm runs on C-sets vs their encoding as typed graphs. We see straightforward relationships of speeds up 2-3x and memory savings in virtue of the C-set representation.

More work is needed before we can try comparing our work directly against nauty.



Performance: future improvements

Ordering objects in schema for early pruning

- Analysis of schema hints at which objects will be resolved quickly by color saturation.
- “∂_1: [2, 1, 3...], ∂_2: ..., ∂_3: ...”
- “src: [1, 3, 4, ...], tgt: ..., ∂_1: ...”

$$\begin{tikzcd}
T \arrow[r, bend left, "\partial_1"] \arrow[r, "\partial_2" description] \arrow[r, bend right, "\partial_3"'] & E \arrow[r, shift left, "src"] \arrow[r, shift right, "tgt"'] & V
\end{tikzcd}$$

Efficient memory layout

- Memory organized by table
- Indexed foreign keys

Use of schema equations to infer symmetries

49

We have reasons to believe a mature implementation of this algorithm would favorably compare against nauty, for problems that have a richly structured schema (not for graphs). One simple reason for this is the memory layout of C-sets (which are represented as databases).

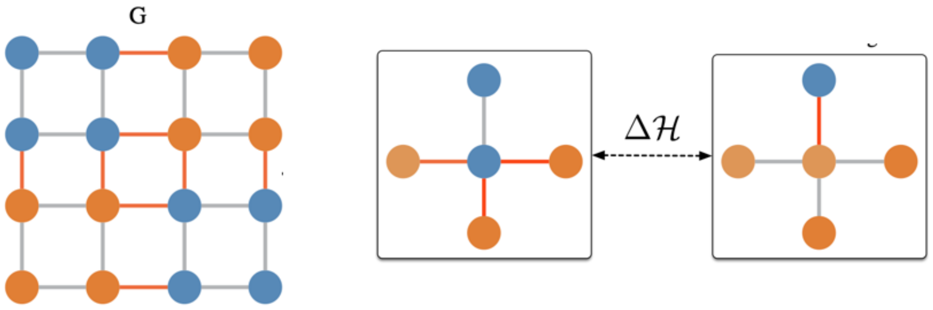
However there are other things we can take advange of too, uniquely for C-sets. We can analyze the schema and set the canonical order in such a way that we can prune branches early.

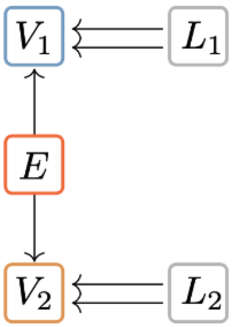
Pressing Questions

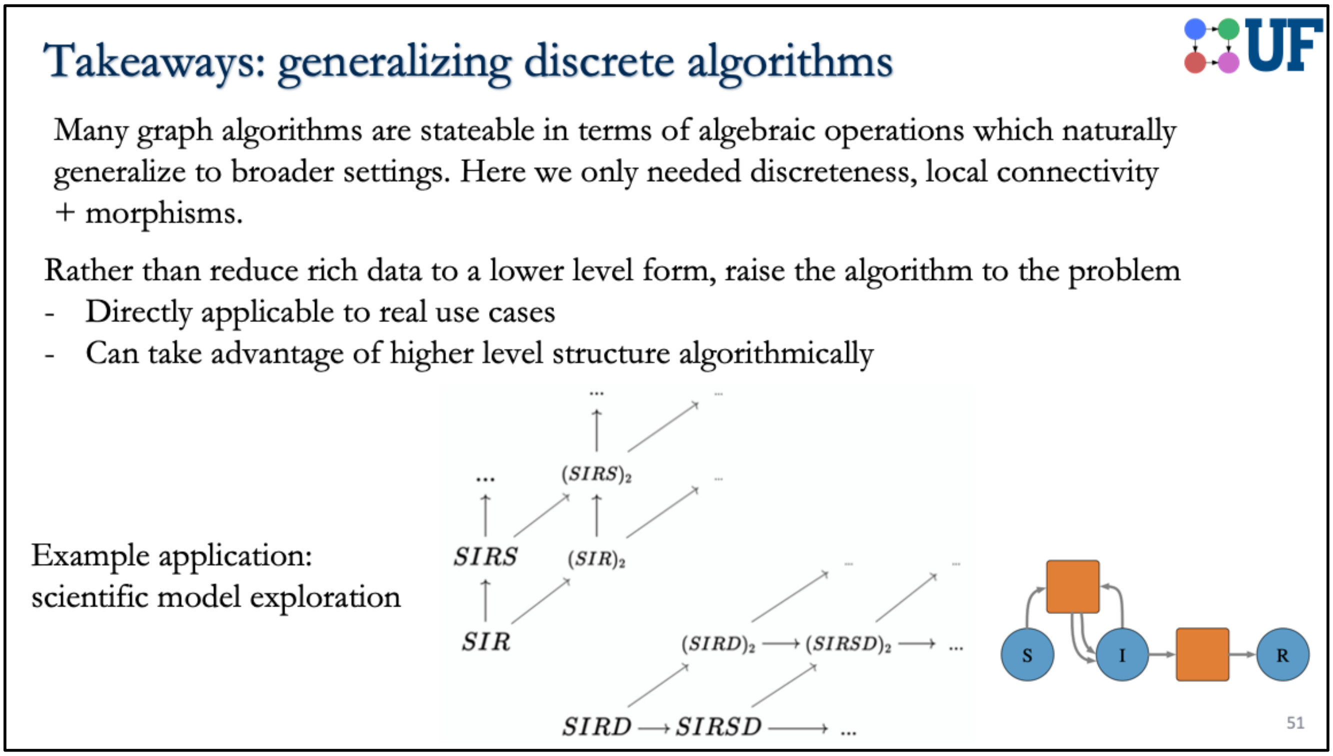
Can we randomly search the space of C-sets up to isomorphism?

Can we search for homomorphisms up to isomorphism?

Ising model MCMC simulation via C-set rewriting







Takeaways: generalizing discrete algorithms

Many graph algorithms are stateable in terms of algebraic operations which naturally generalize to broader settings. Here we only needed discreteness, local connectivity + morphisms.

Rather than reduce rich data to a lower level form, raise the algorithm to the problem
- Directly applicable to real use cases
- Can take advantage of higher level structure algorithmically

Example application: scientific model exploration

51

one of the main takeaways we get from this project is that, while it's possible to reduce one's problem to a simple form, such as a graph for nauty, or perhaps some rich higher order logical formula which si reduced to first order logic so that a well-optimized solver can handle. An alternative is to generalize the algorithm itself, which has the potential to be faster by taking advantage of higher-level structure that gets lost when you squash your problem down to the simple format.

We've already found great use for this algorithm in the case of scientific model explroation, where for example, epidemiologists may be exploring possible chemical reaction networks to represent the dynamics of some disease, that this is a branching search process that may hit upon repeats. It's important to be able to work with these scientific models up to isomorphism, to reduce the branching factor.

Thanks!