Implementing Decision Diagrams in Production Systems

DPSOLVE 2023

👋 This is a talk about 🐇 and 🐇🕳️

Ryan O'Neil

CTO at Nextmv

Integer scientist, cat and early music enthusiast, Go programmer

🏃 Speedrun
Let's see some Decision Diagrams in the wild!

🤔 Why?
Decision Diagrams have unique characteristics.

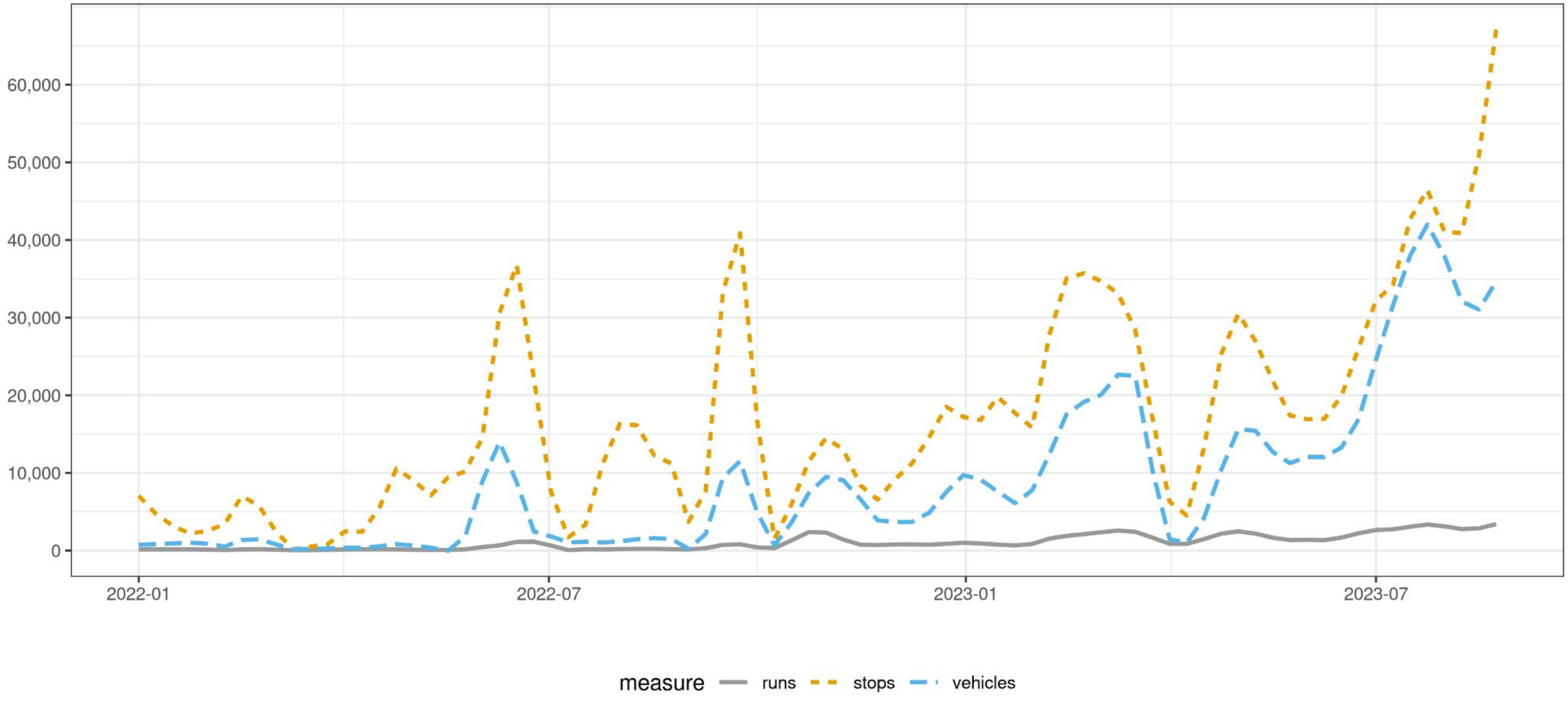
🛠️ How?
How this work and some things we learned building it.

🤷 Q&A time
You probably have questions. I know I do.

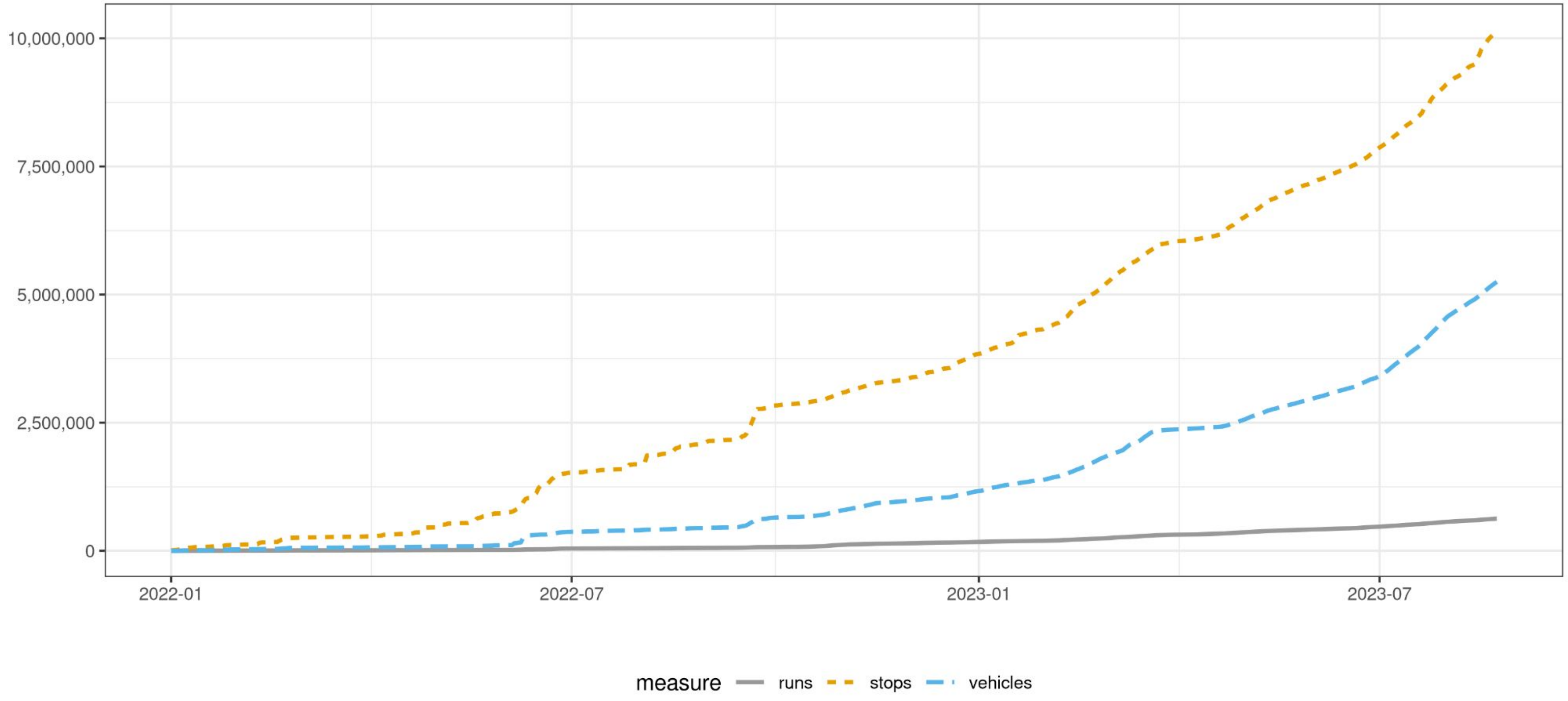
🏃🏃🏃

Speedrun

📈 Daily hosted routing metrics



📈 Cumulative hosted routing



🤔🤔🤔

Why?

🍳 Dynamic meal delivery

Original motivation came from routing at Zoomer.
Work continued at Grubhub Delivery.

Both solved dynamic meal delivery problems.
The biggest difference was scale.

We solve lots of routing (and other) problems at Nextmv.
We've had the fortune to test out DDs on some of them!

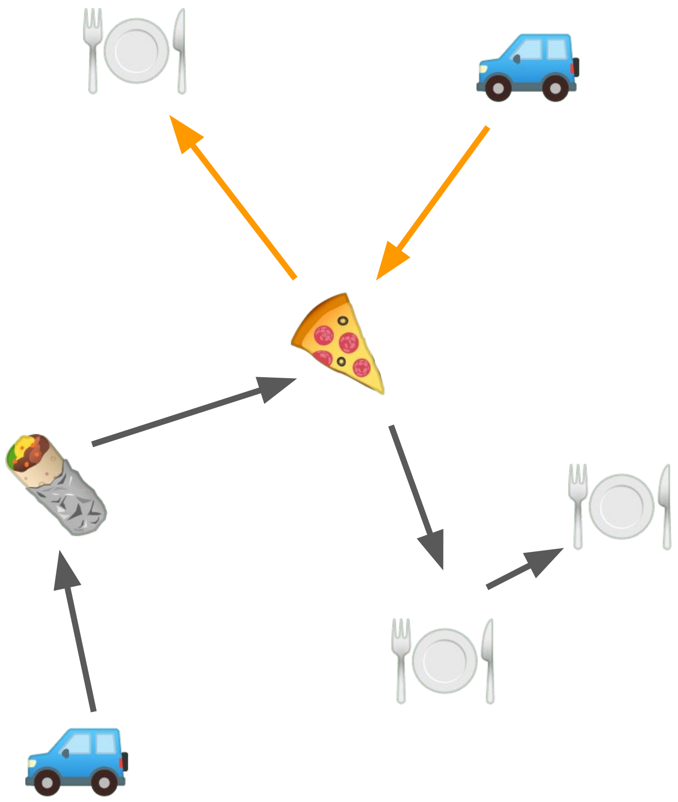
🍳 Dynamic meal delivery

Orders arrive dynamically throughout the day.

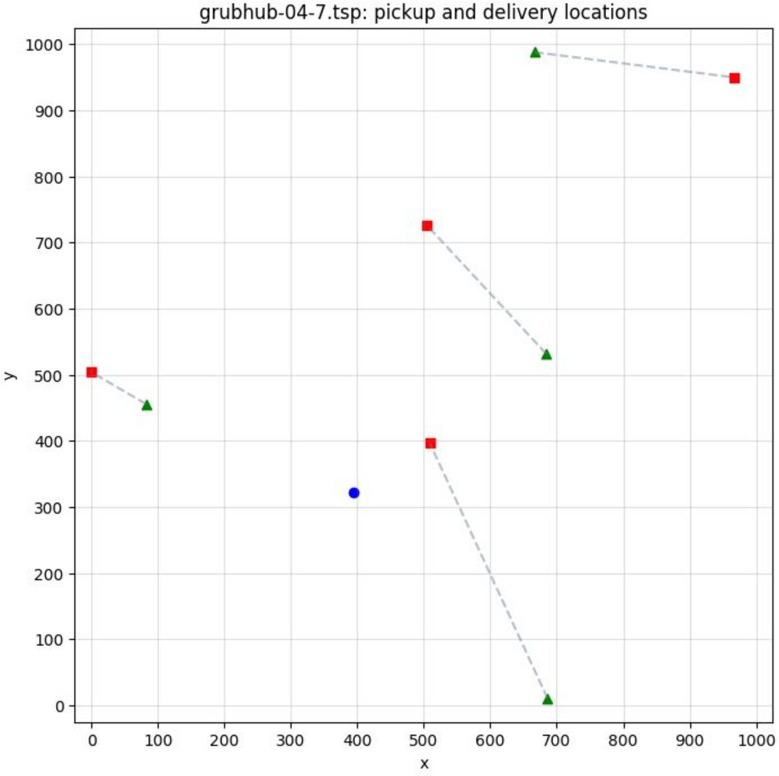
A **shared driver pool** serves many restaurants.

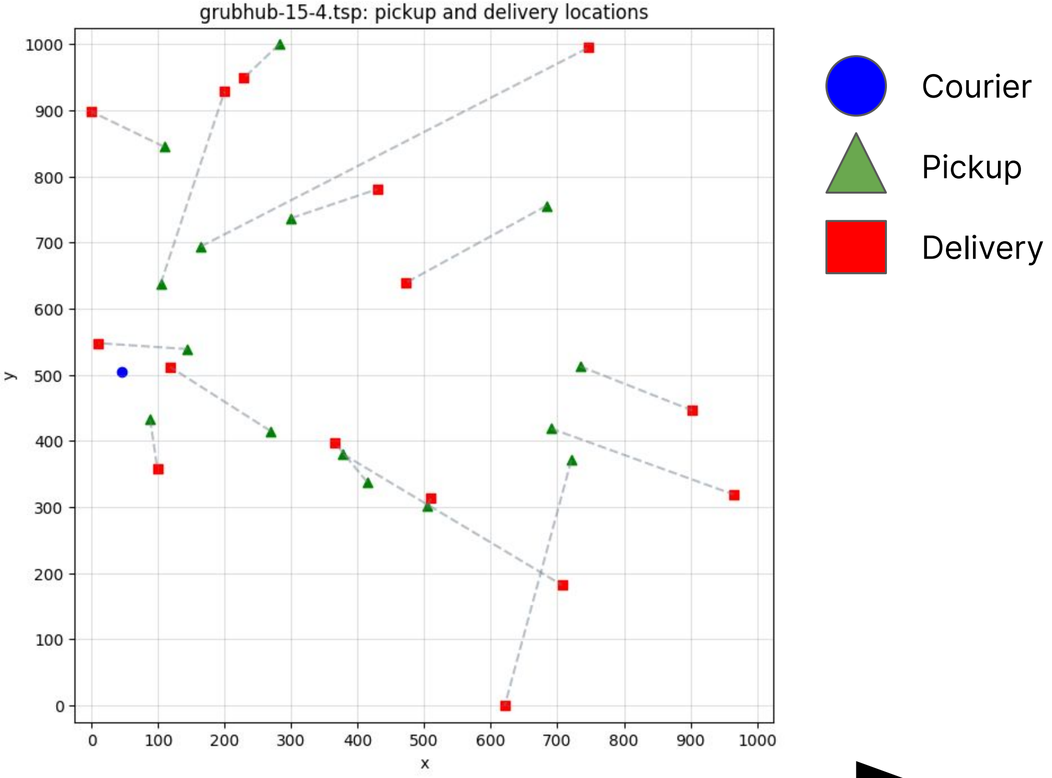
Multiple orders are consolidated for efficiency.

Problems **get large** (thousands of orders, hundreds of drivers).



On-demand last-mile is everywhere now

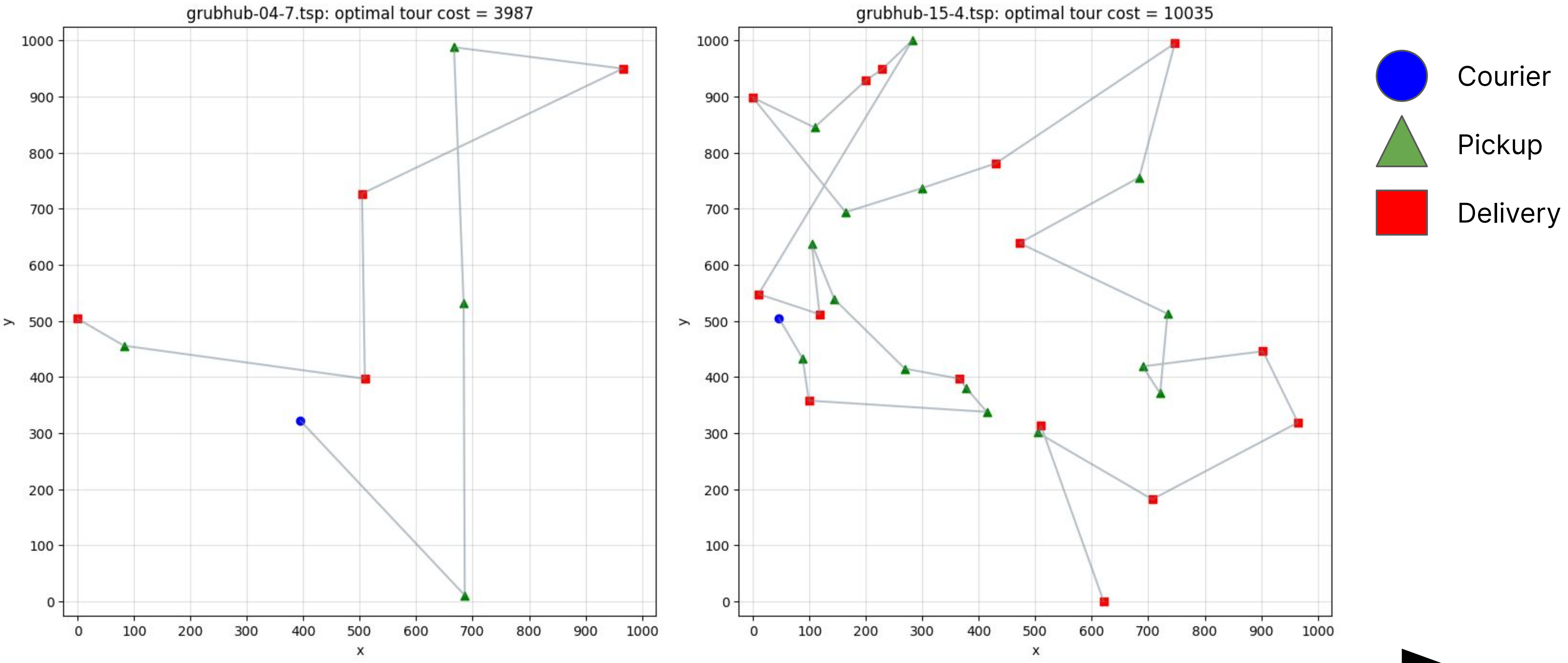




People, Meals, Perishable Goods

Groceries, Packages, Non-Perishable Goods

🤳 And speed to operational solutions is important

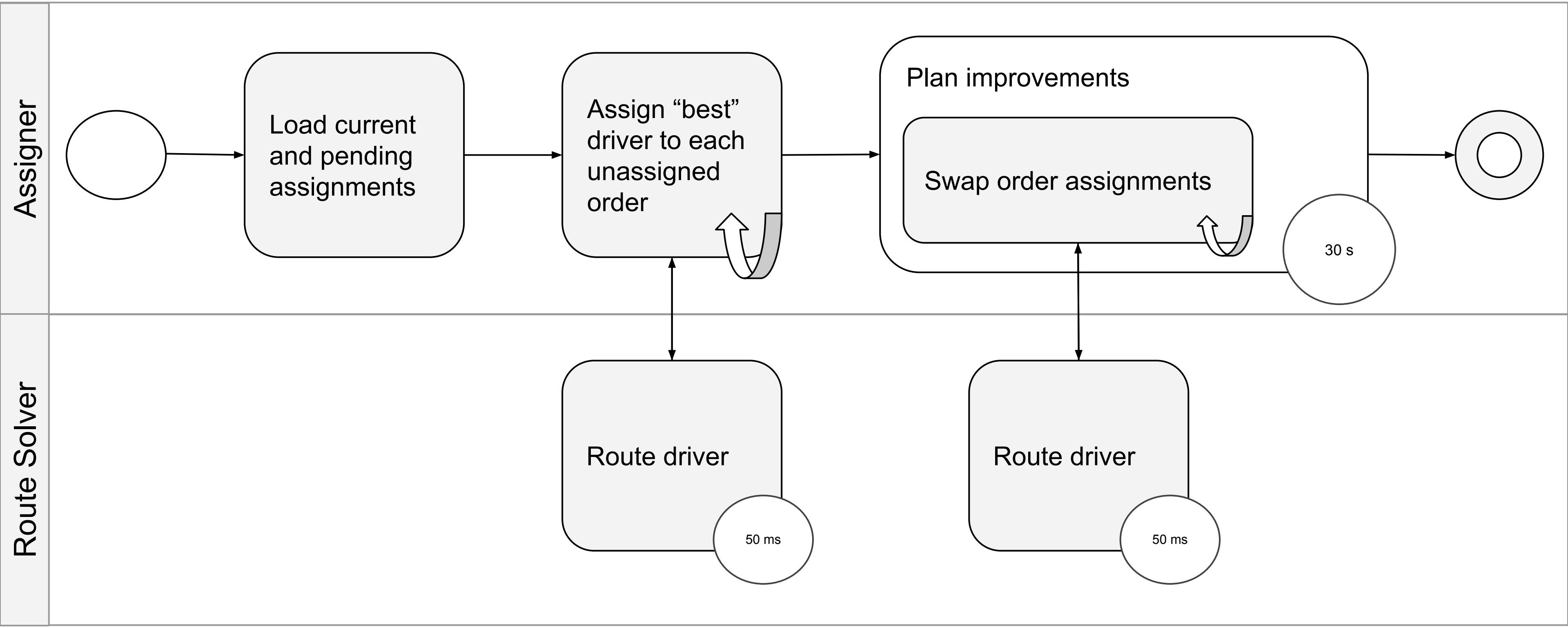


n

People, Meals, Perishable Goods

Groceries, Packages, Non-Perishable Goods

🍳 Real-time planning often looks like this

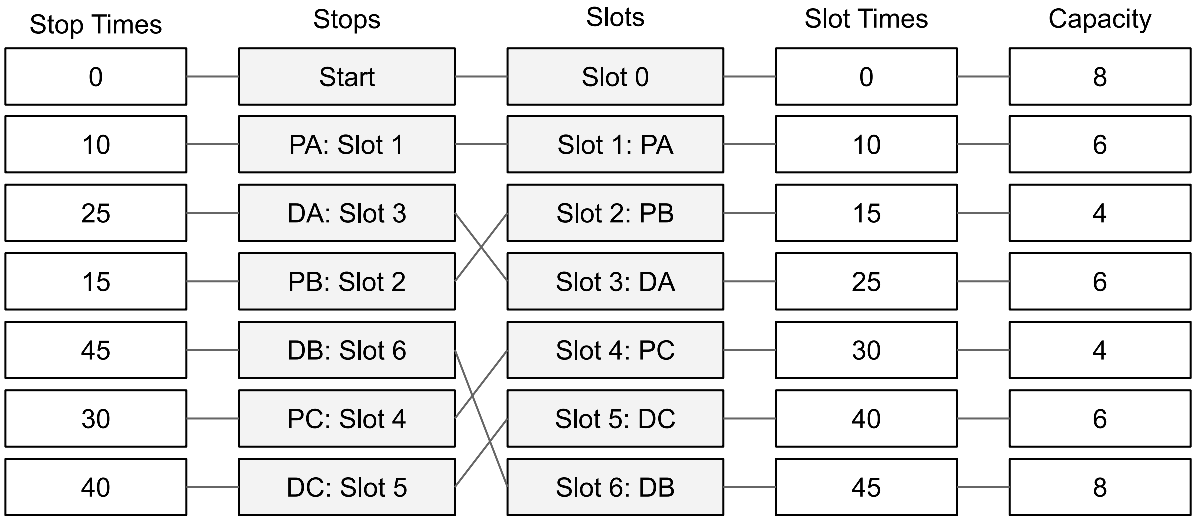


Replan every 30s to 2m for real-time operations.

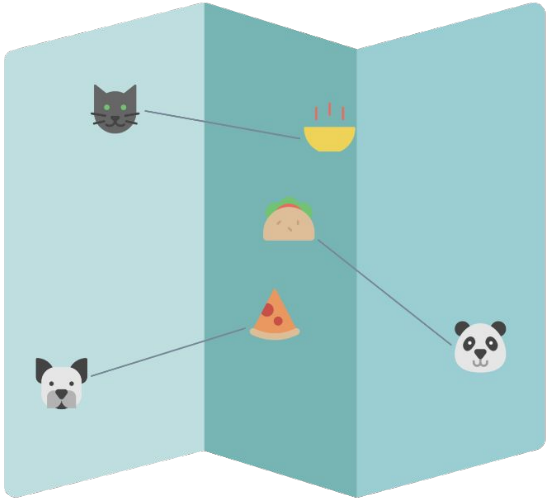
🤯 Models can get pretty crazy looking

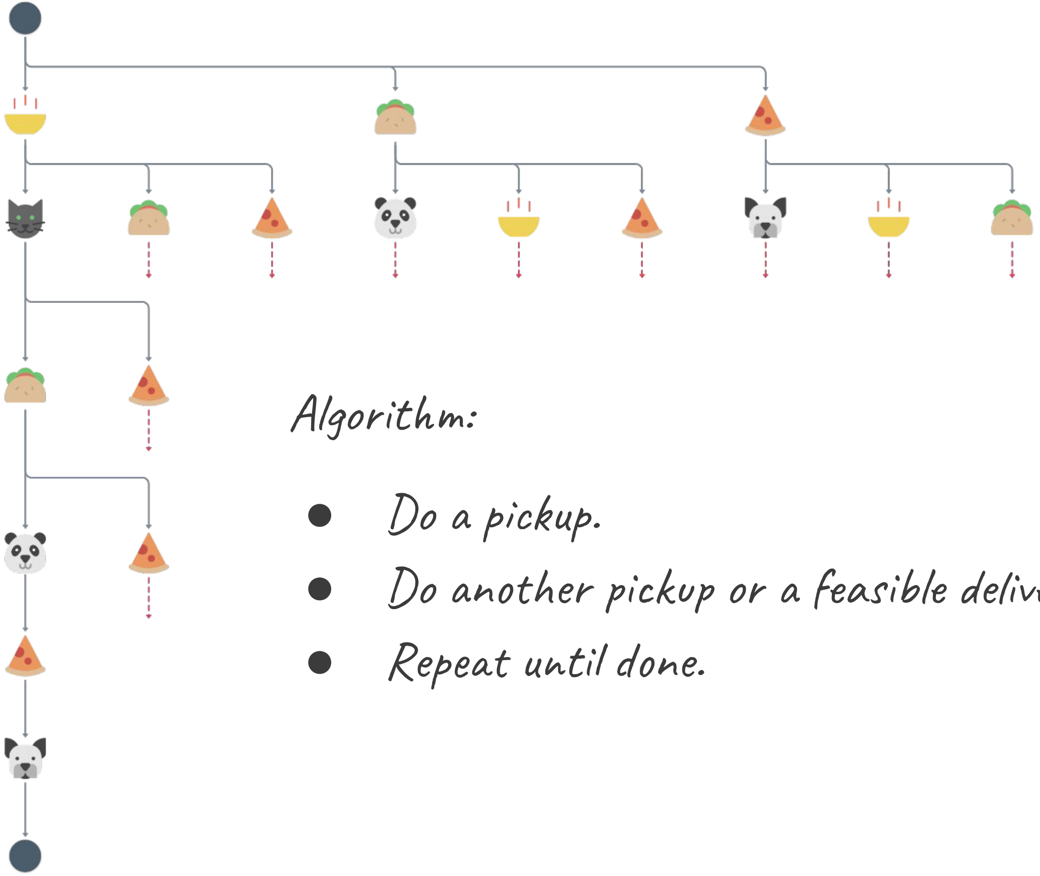
Adjacency Matrix (column major)

	S	PA	DA	PB	DB	PC	DC
S							
PA	X						
DA				X			
PB		X					
DB							X
PC			X				
DC						X	



🙂 In contrast, DDs can seem pretty simple...

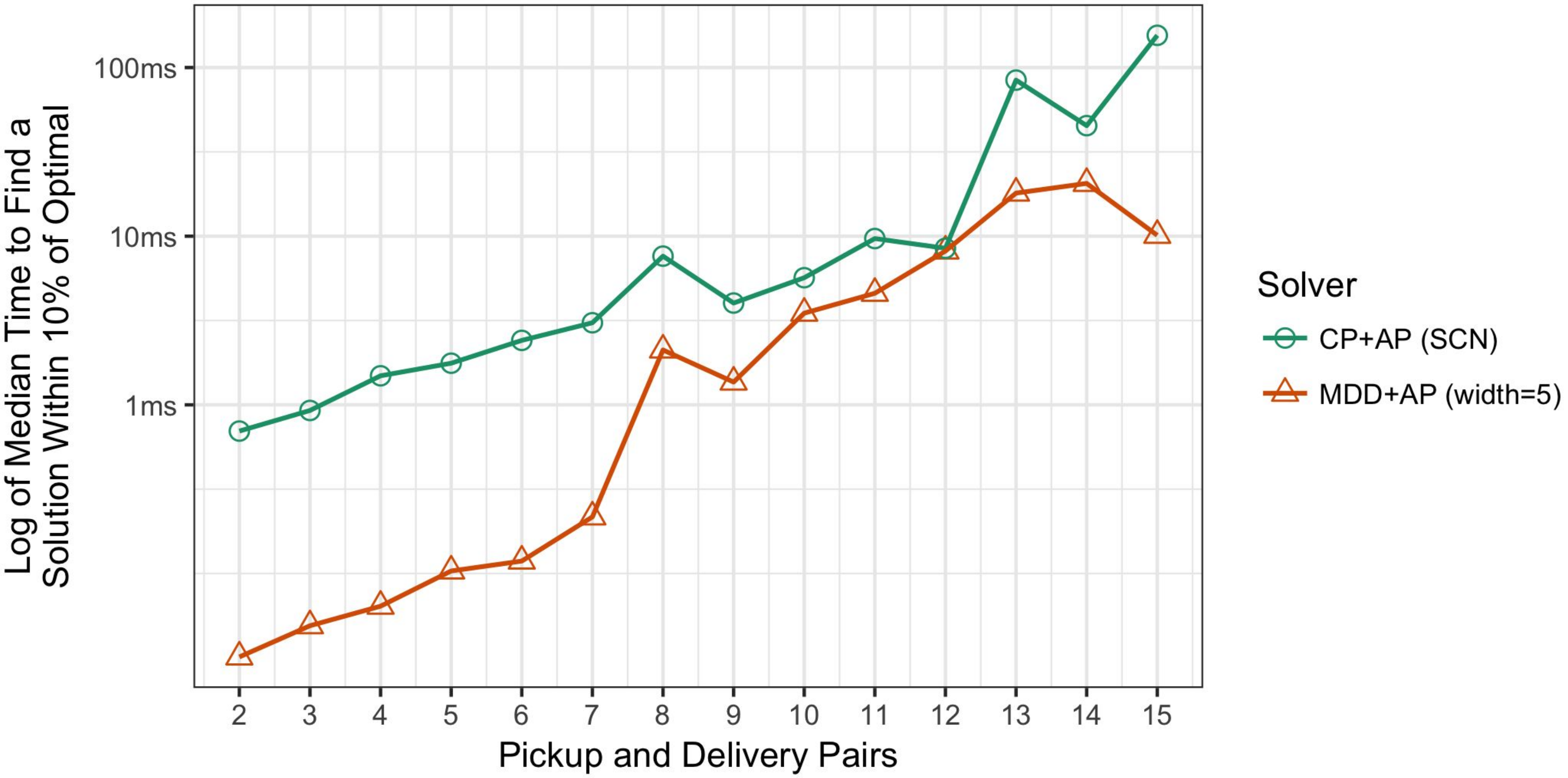




Algorithm:

- *Do a pickup.*
- *Do another pickup or a feasible delivery.*
- *Repeat until done.*

🏇 In the right hands, they can even be fast!



🧐 In the now times for decision and OR ops

Our <u>data</u> looks more like this:

```
            "id": "vehicle-10"
            "capacity": 125
        }
    ],
    "stops": [
        {
            "id": "order-1-pickup-1"
            "position": {
                "lon": -96.827094
                "lat": 33.004745
            },
            "precedes": "order-1-dropoff",
            "quantity": -27
        },
        {
            "id": "order-1-dropoff"
            "target_time": "2023-05-25T04:24:20-6:00"
```

But our <u>process</u> looks like this:

- Translate business rules to linear inequality systems
- Hand off to a solver
- 🤞🙏
- Translate solutions back to business rules

So if we think about optimization as a tool for solving operational problems on operational data…

…can we build models in a way that's more natural to the problem we're trying to solve?

Where do DDs live in the world of optimization?

MIP

Mixed integer programming

Strong optimality reasoning

"This is the best solution!"

DDs

Decision diagrams

Good at finding feasible solutions, can prove optimality

"This is a good, timely solution!"

CP

Constraint programming

Strong feasibility reasoning

"Here are solutions!"

Somewhere in the middle

🛠️🛠️🛠️

How?

🏆 A solver is an oracle in software…

…most software reads, manipulates, and writes **state data**…

…so how can we tell a solver about the data structures we want, and have it **fill in the details?**

📖 Hop is a Decision Diagram solver(-ish)

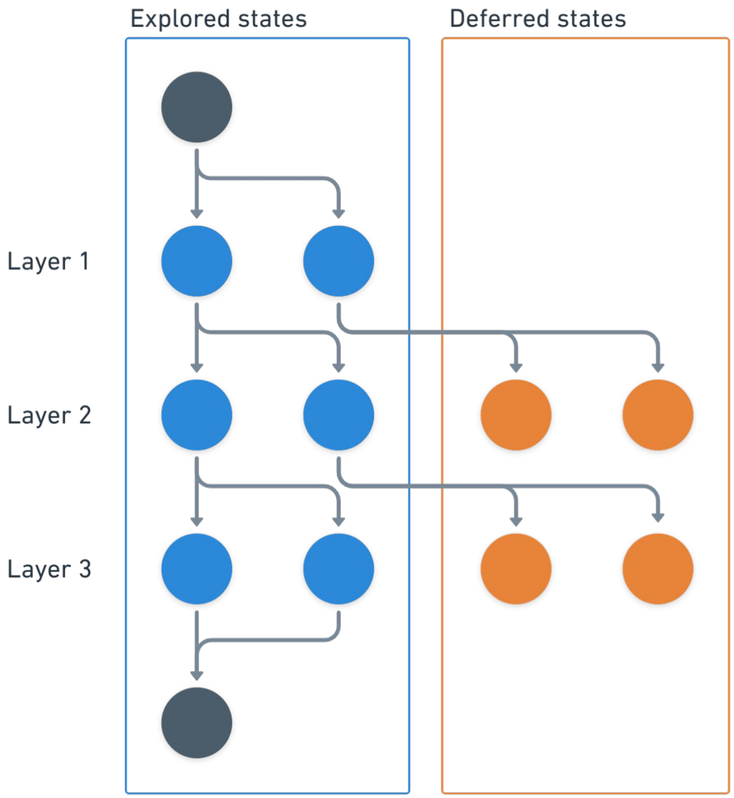
- Hop executes a branch-and-bound over states composed of **arbitrary state data**.
- Instead of relaxation diagrams and merge operators, Hop relies on **state expansion**.
- Hop supports problem-specific **top-down reduction**.
- Hop assumes state **data is immutable**.

👀 Hop’s explores rectangles

SEARCH

Restricted diagram

Selectively explore some states now and some states later

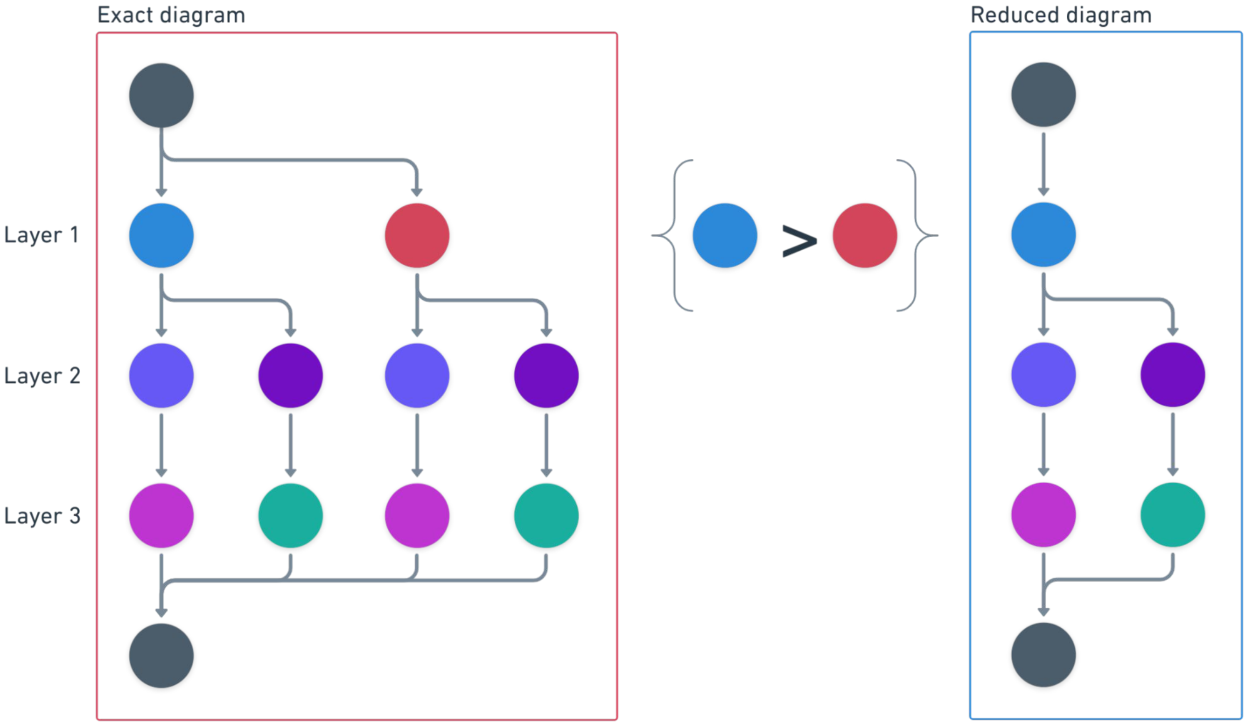


👀 Reduce diagrams as we go

INFERENCE

Reduced diagram

Learn as we explore to avoid unproductive branches of the search tree.

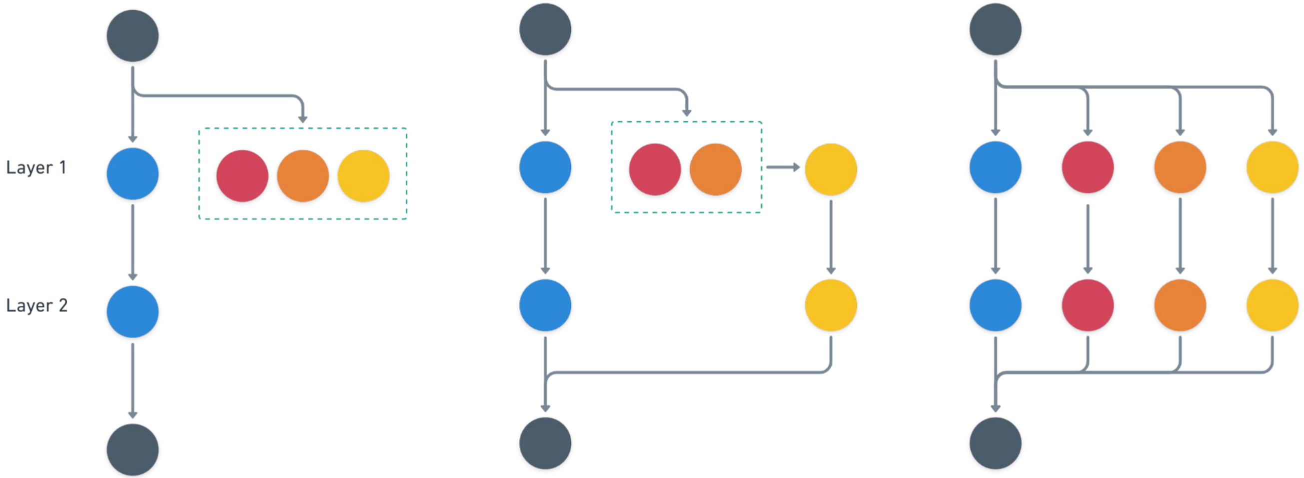


👀 Expand states on demand

RELAXATION

State expansion

Create new exact states as requested by the search. This avoids merging wide layers.



State interface

Any state can be feasible, not just the terminal node.

For example: a 0-1 knapsack state is feasible if `weight ≤ capacity`.

Values need not increase or decrease monotonically, or be within bounds.

Bounds only tighten in child states.

```go
package model

import "context"

type State interface {
    Feasible() bool
    Next(ctx context.Context) Expander
}

type Valuer interface {
    State
    Value() int
}

type Bounder interface {
    Valuer
    Bounds() Bounds
}
```

Expander interface

An expander generates new states when requested.

Diagram options include an expansion limit.

Expansion can be eager or lazy.

```
package model

type Expander interface {
	Expand() (State, bool)
}
```

Solver interface

A solver sends improving (all) feasible solutions over a channel as it finds them, or the best (last) solution it finds.

A solution is a state with metadata.

Some solvers can also infer. Mosty this is used for bounds messaging.

```go
type Solver interface {
    All(ctx context.Context) <-chan Solution
    Last(ctx context.Context) Solution
    Options() Options
}

type Inferrer interface {
    Solver
    Infer(context.Context, model.Valuer)
}
```

Options

Options control diagram creation (reduction, restriction, expansion).

They also control queue discipline, selection of deferred nodes in search, and termination criteria.

```go
type Options struct {
    Sense   model.Sense
    Tags    map[string]any
    Diagram struct {
        Reducer    reduce.Reducer
        Restrictor restrict.Restrictor
        Width      int
        Expansion  struct {
            Limit int
        }
    }
    Search struct {
        Queuer   queue.Queuer
        Searcher search.Searcher
        Buffer   int
    }
    Limits struct {
        Duration  time.Duration
        Nodes     int
        Solutions int
    }
    Random struct {
        Seed int64
    }
    Pool struct {
        Size int
    }
}
```

⚙️ High-level modeling interfaces = “engines”

```
route.NewRouter(
    stops,
    vehicles,
    route.Starts(starts),
    route.Ends(ends),
    route.Capacity(quantities, capacities),
    route.Windows(windows, shifts),
    route.Precedence(precedences),
)
```

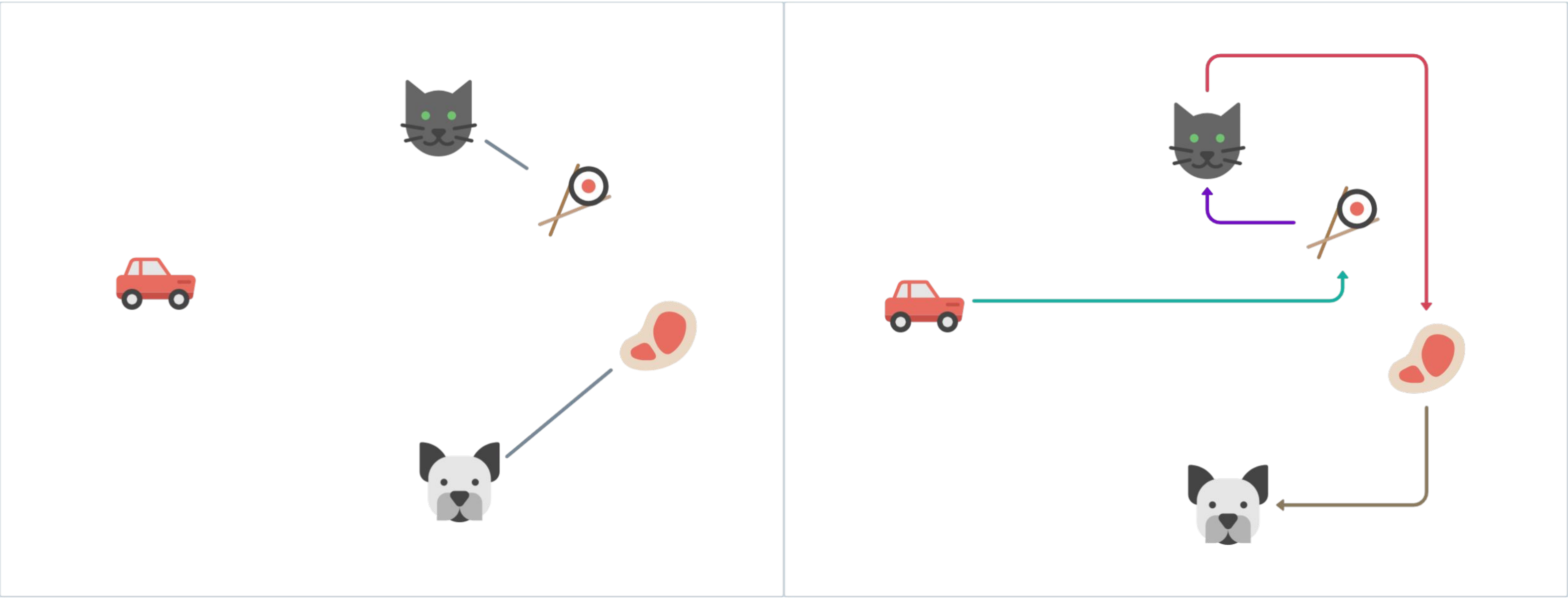
Let’s look at some models.

Hop doesn’t have a modeling language *(yet)*. We’re still learning what that should look like.

Let’s dive a bit deeper into the routing model we saw earlier.

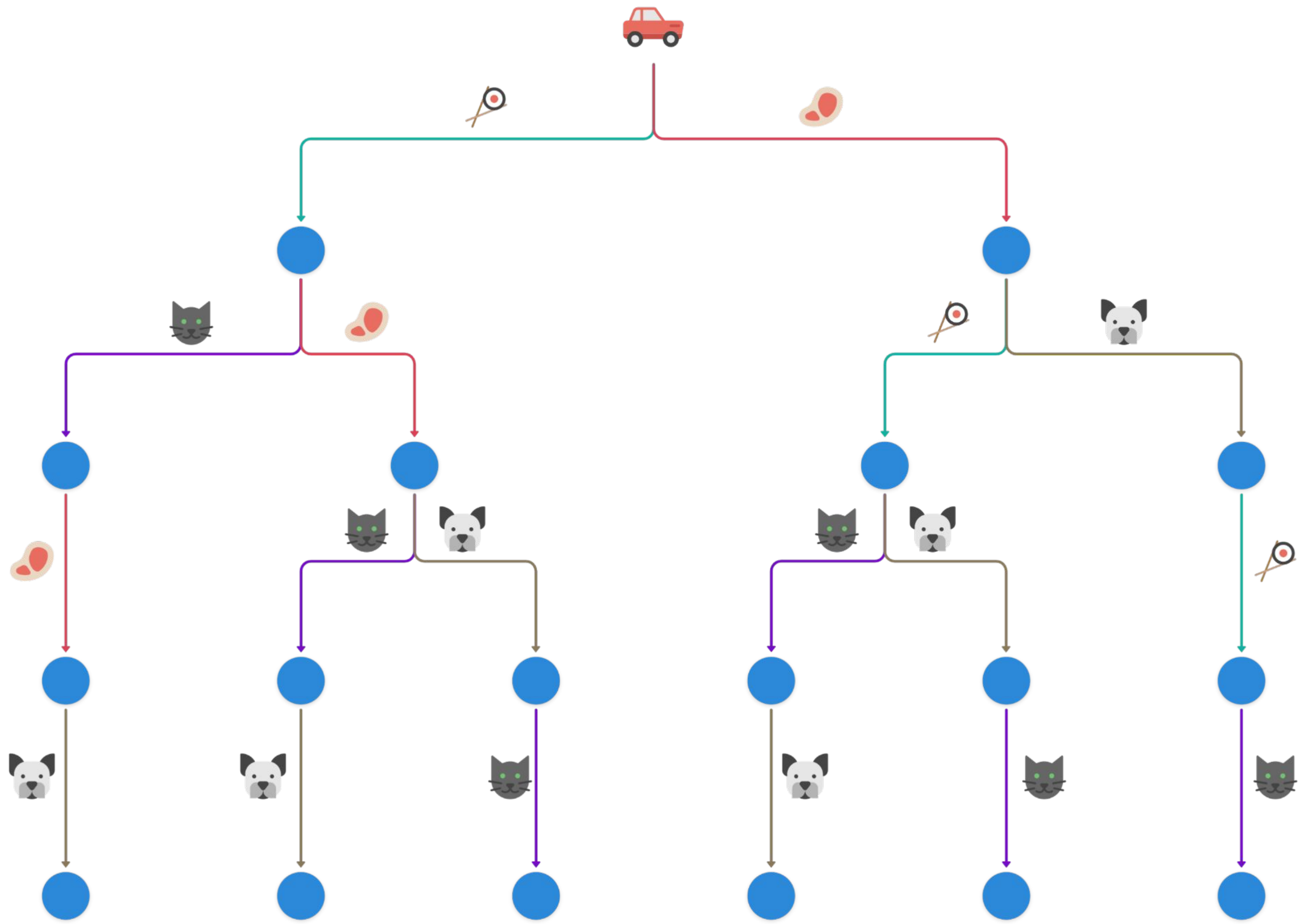
We’ll start with single-vehicle pickup and delivery, then generalize it to multi-vehicle.

😅 An extremely simple example



📃 First we'll look at the "append" model

- This model is a very simple MDD.
- We start at the driver's location.
- At each layer, we try appending all feasible stops.
- Our transition function accounts for precedence, capacity, time windows, and other side constraints.
- Let's look at and exact version of the model...

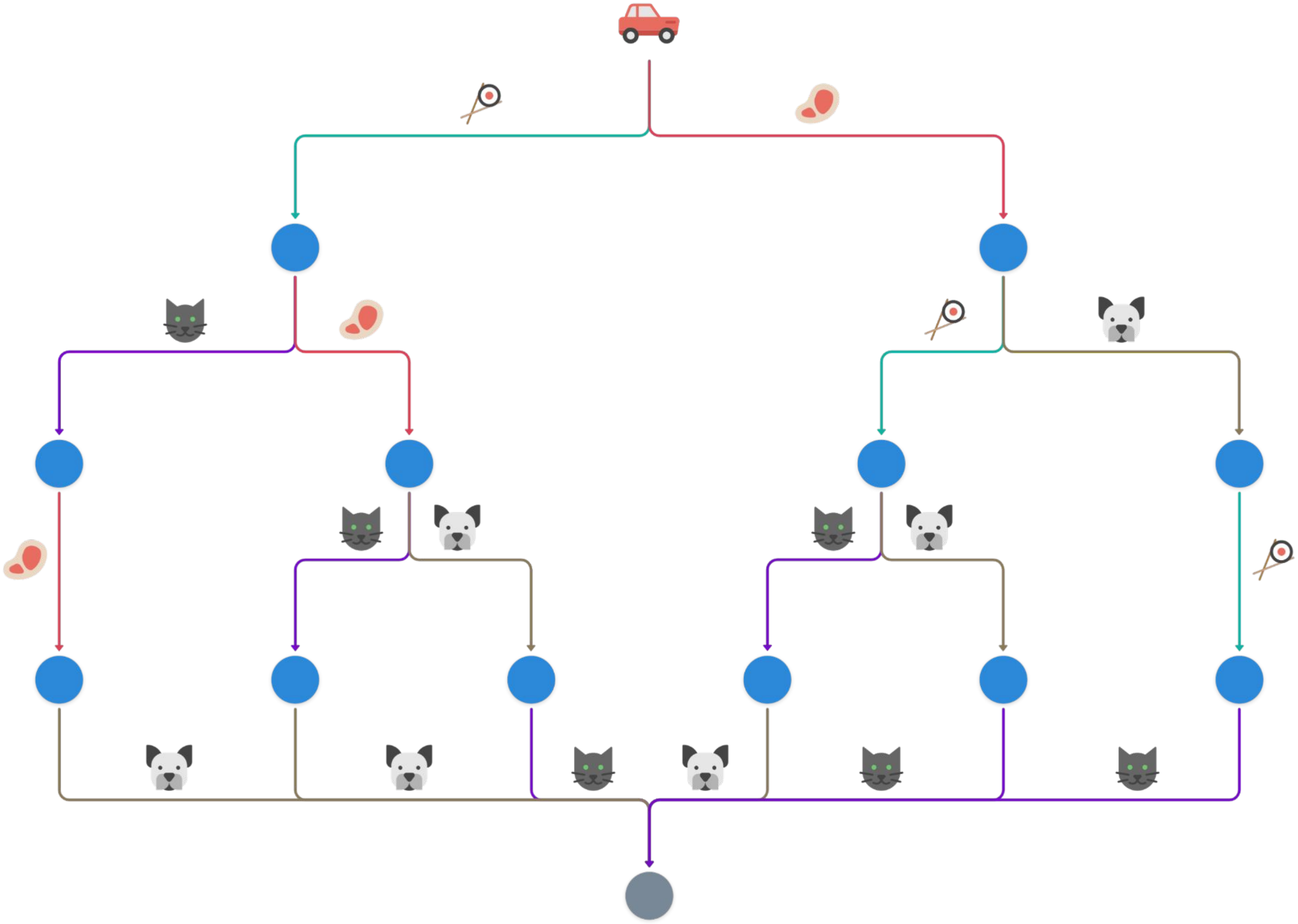


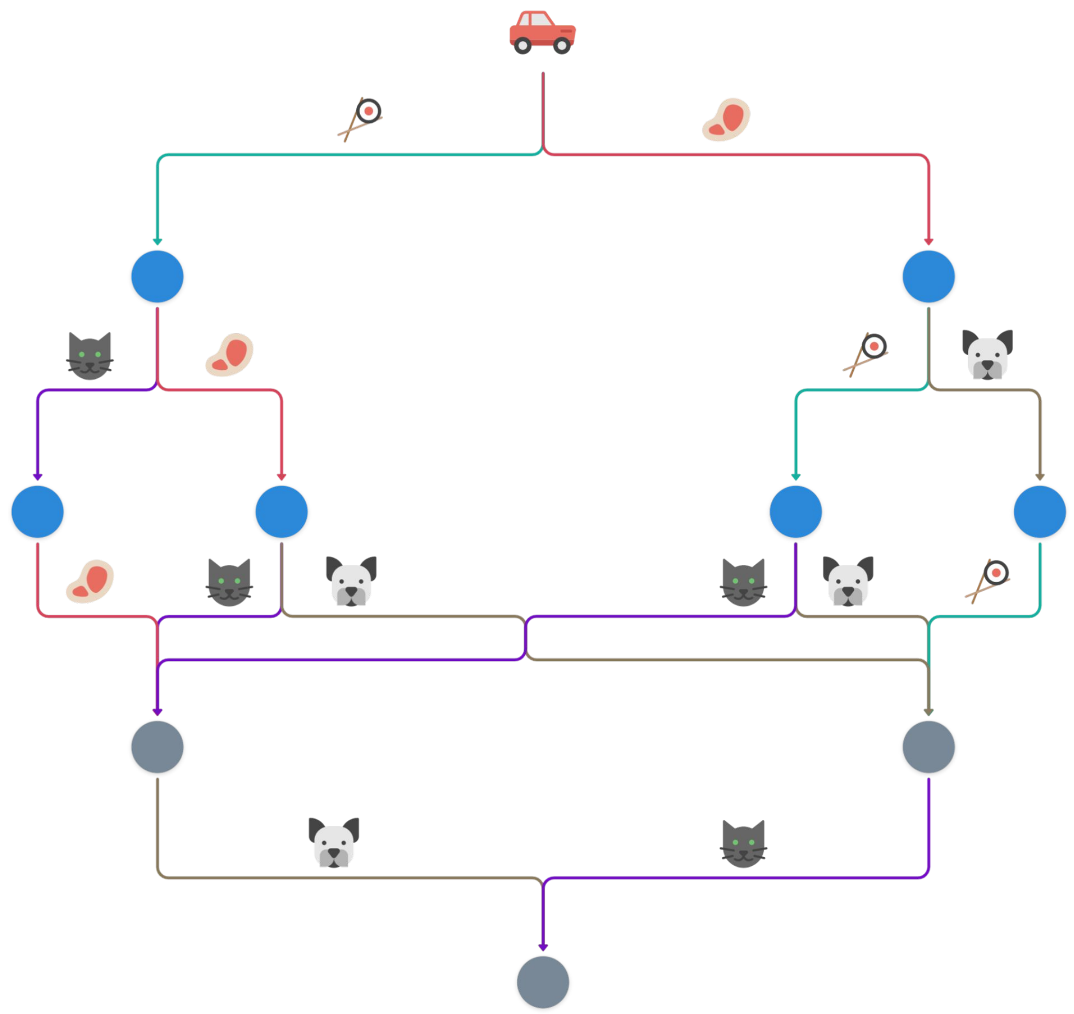
Can this be reduced?

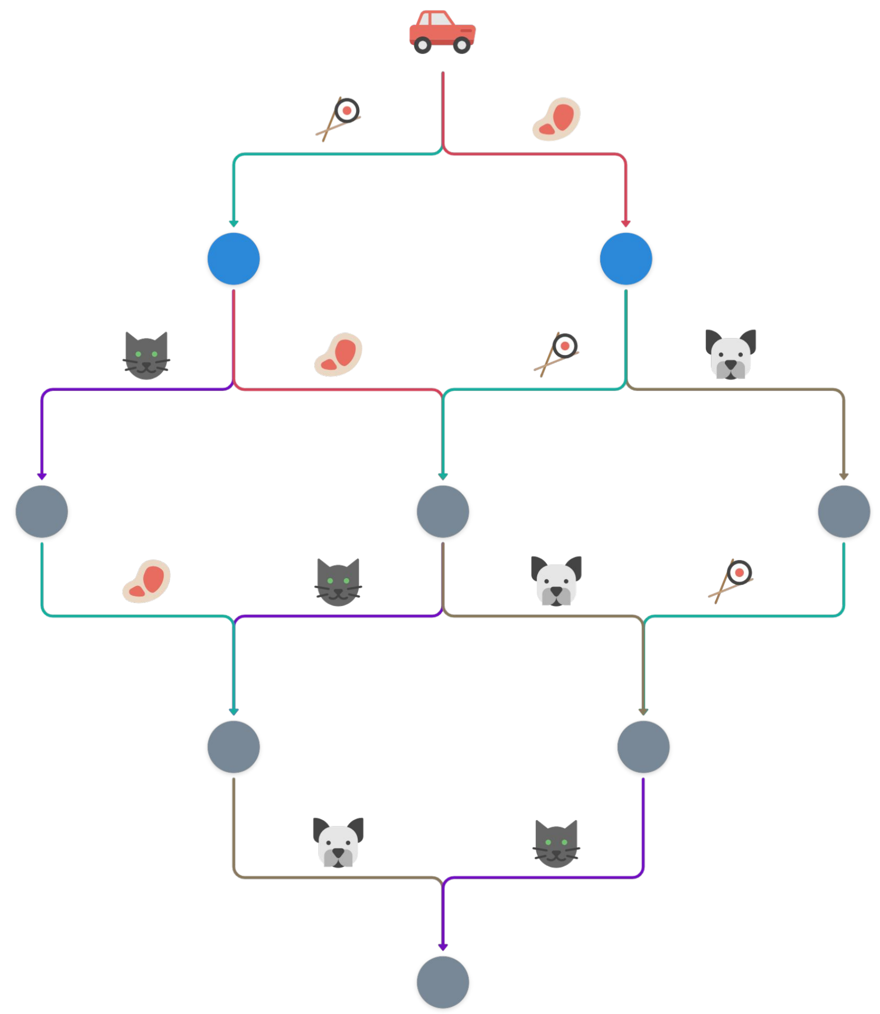
What does this exact diagram look like if we apply a standard reduction technique.

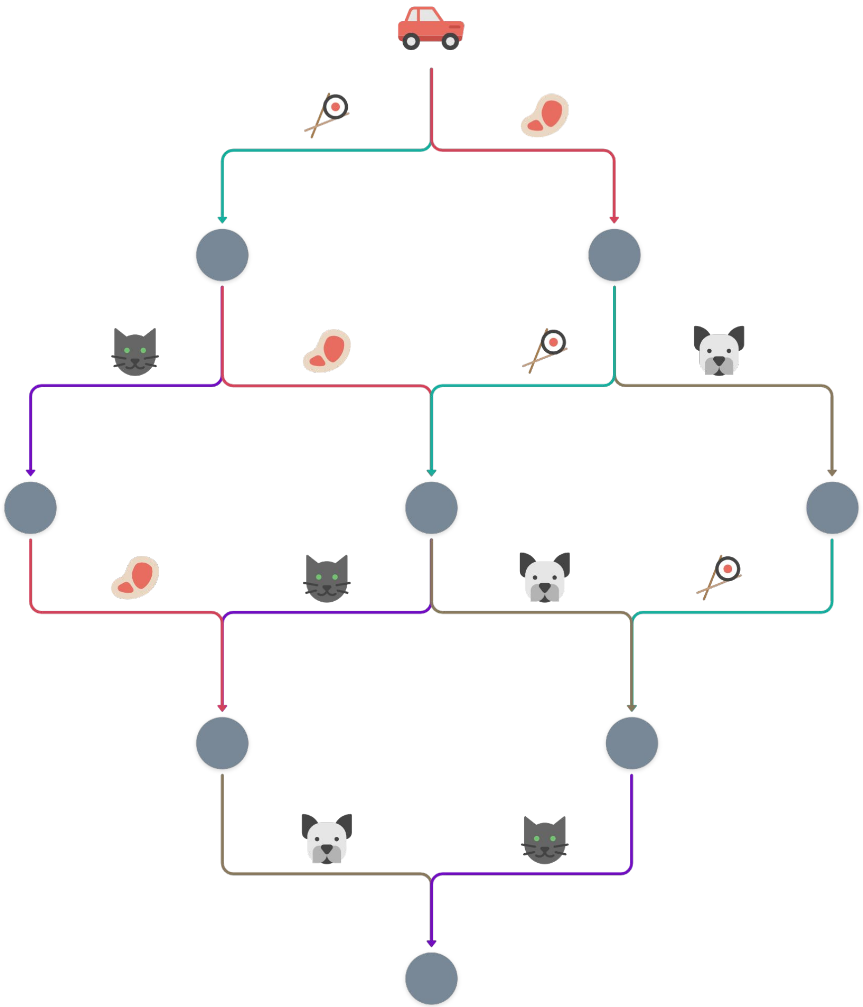
Let’s walk through the procedure.

The gray nodes will be reduced.



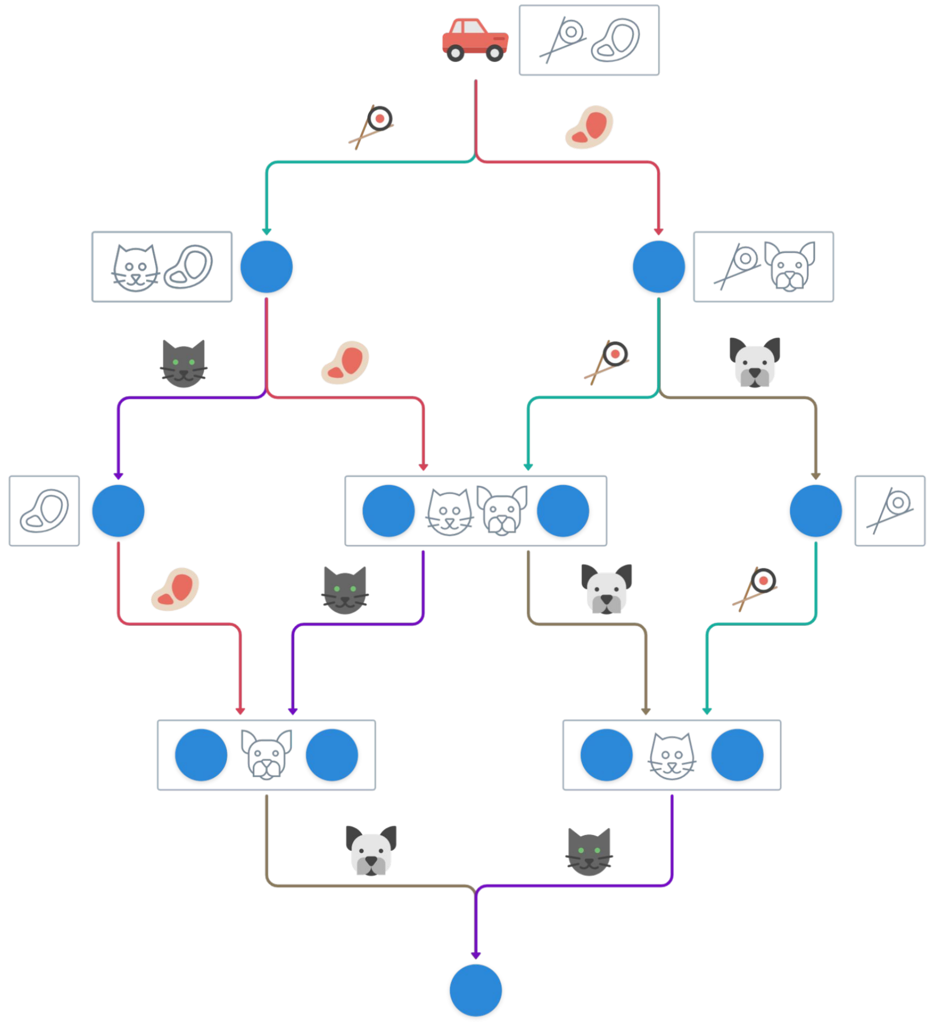






👌 That’s all well and good

- Bottoms-up reduction generates a lot of states we don’t always need.
- This is great for compressing solutions, but in this case I’d prefer not to generate anything I don’t need.
- Is there any way I can reduce the diagram **while I’m constructing it**?

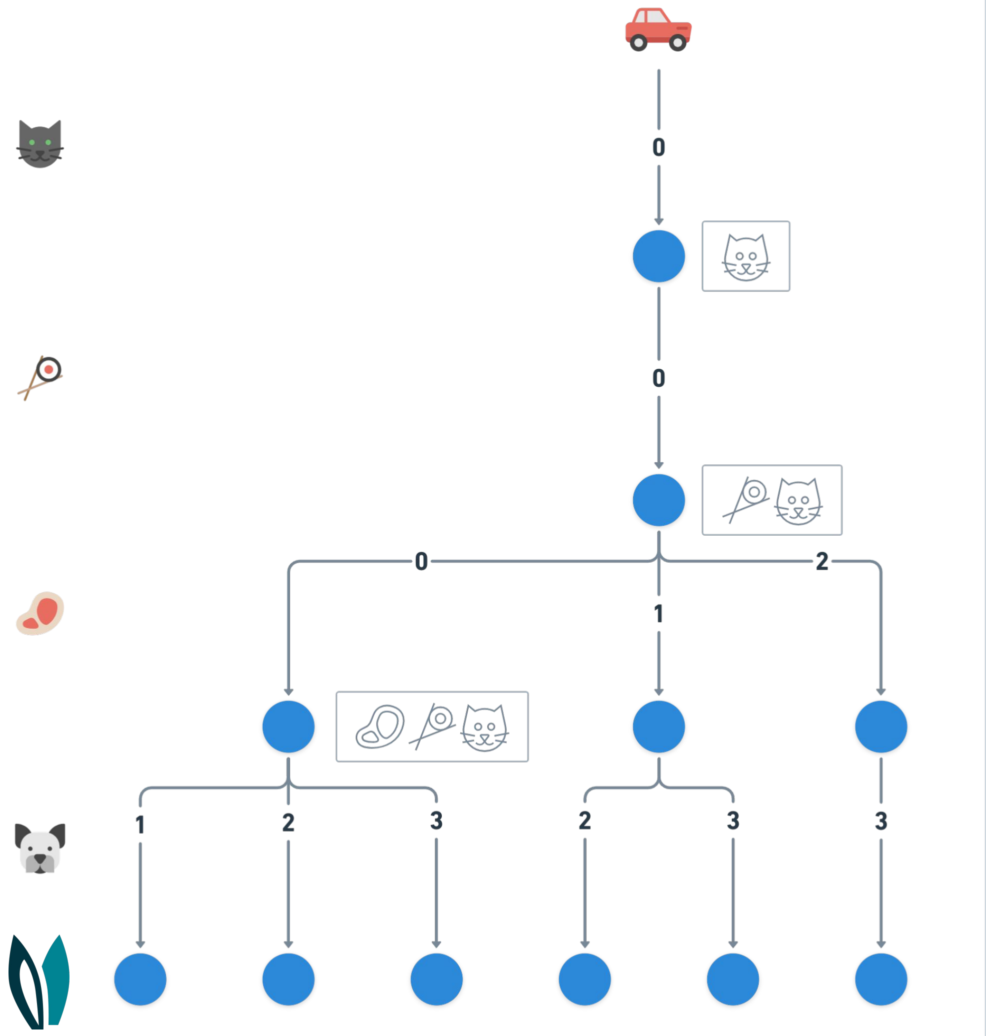
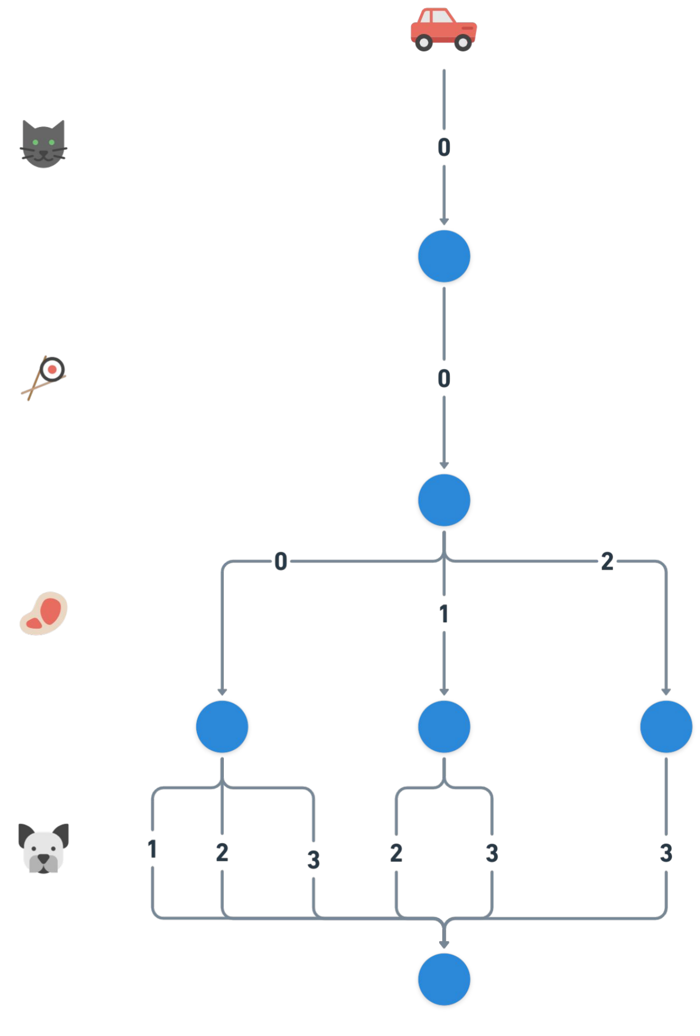


😔 There are gotchas here too

- Now I need a domain store...
- Applying this to arbitrary state data requires custom logic for every model.
- There's a subtle issue with diagram width here too.

➕ Now let's look at the "insert" model

- This model is a bit more complex.
- Order the stops *somehow* (a greedy heuristic works well).
- Start with an empty route: [ ]
- Each layer asks, "at which index do we insert this stop?"
- Again, the transition function accounts for precedence, capacity, time windows, and other side constraints.

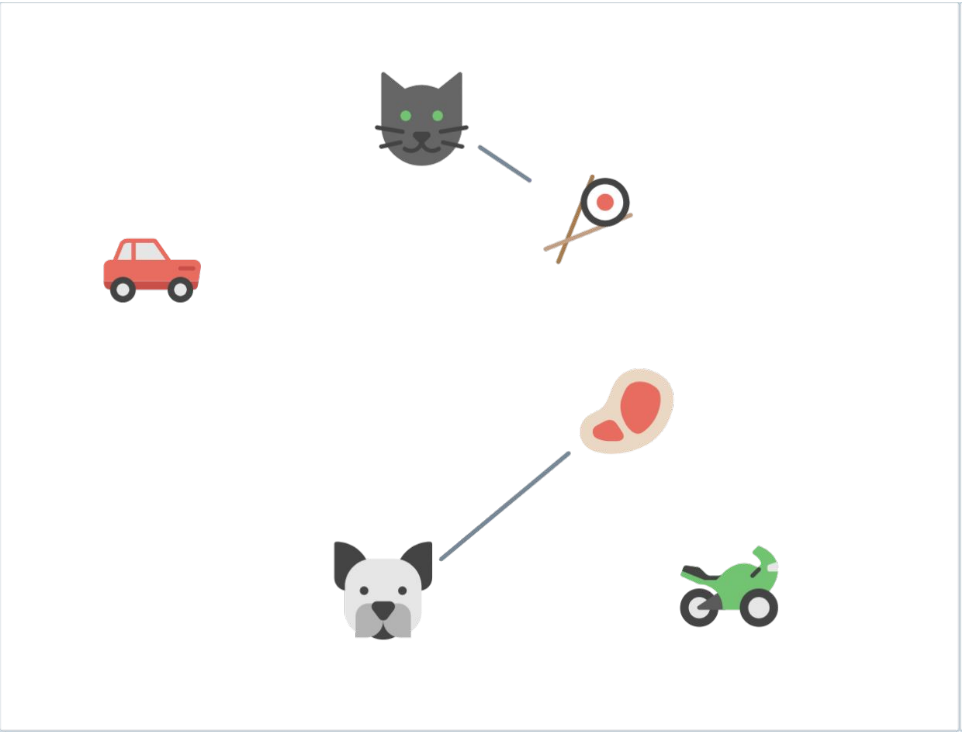

0
0
0
1
2
1
2
3
2
3
3

0
0
0
1
2
1
2
3
2
3
3

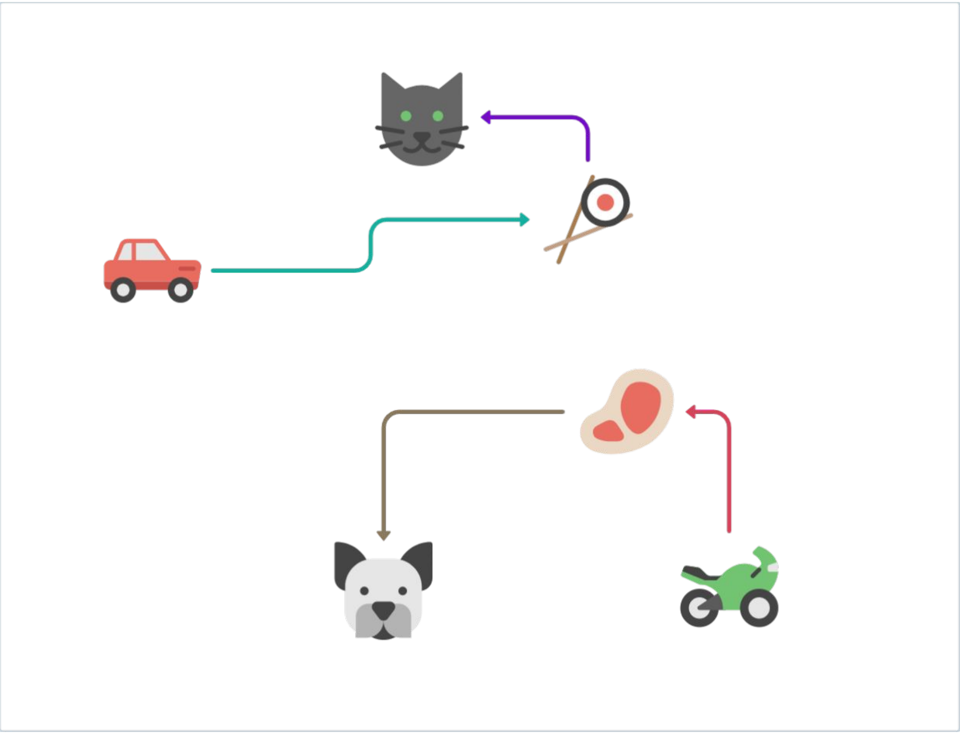
💪 Some strengths of the different models

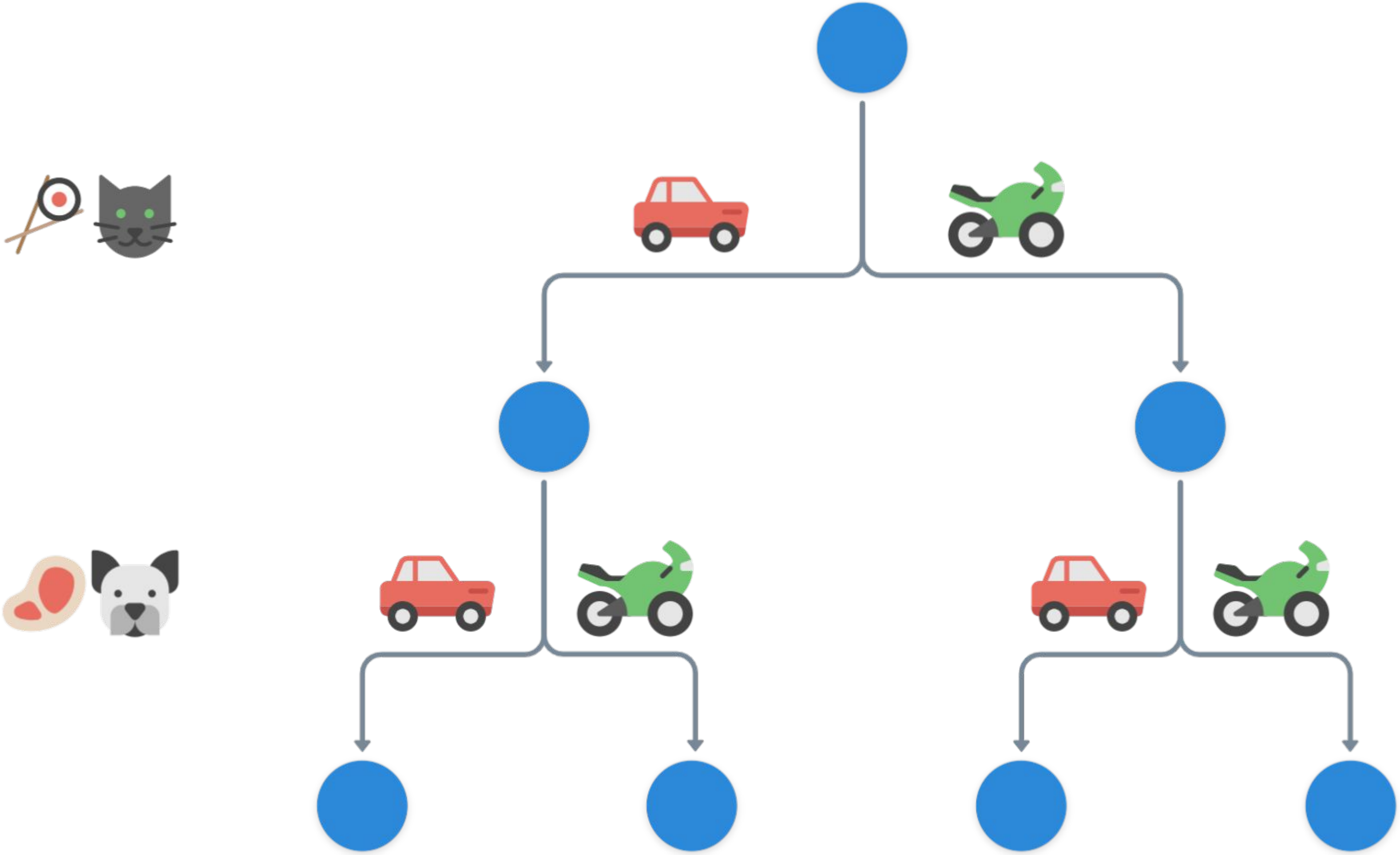
	Append	Insert
Speed to solution quality		✅
Memory efficiency	✅	
Can operate on partial states		✅
Simplicity of implementation	✅	
Side constraint simplicity	✅	
The hard stuff: synchronization, handoff, containment	✅	

🌎 How do we extend this to a fleet?

- We can also decouple assignment and routing, either in a single diagram or in multiple.
- Layers correspond to groups of stops that go together.
- Arcs decide which vehicle we assign them to.



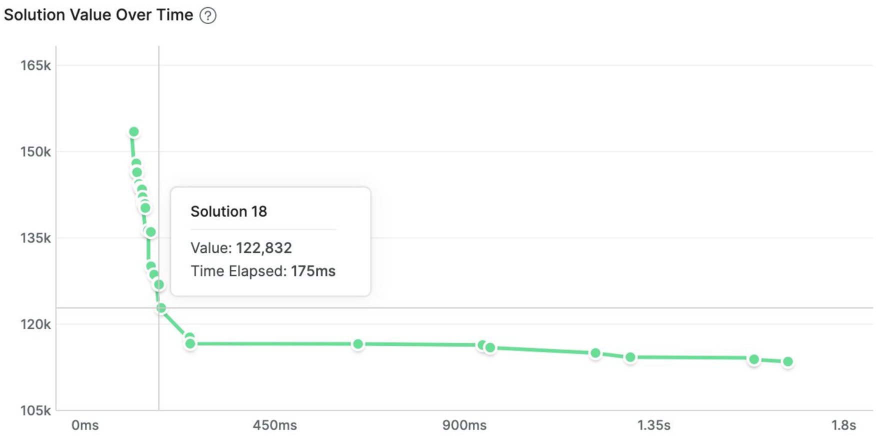


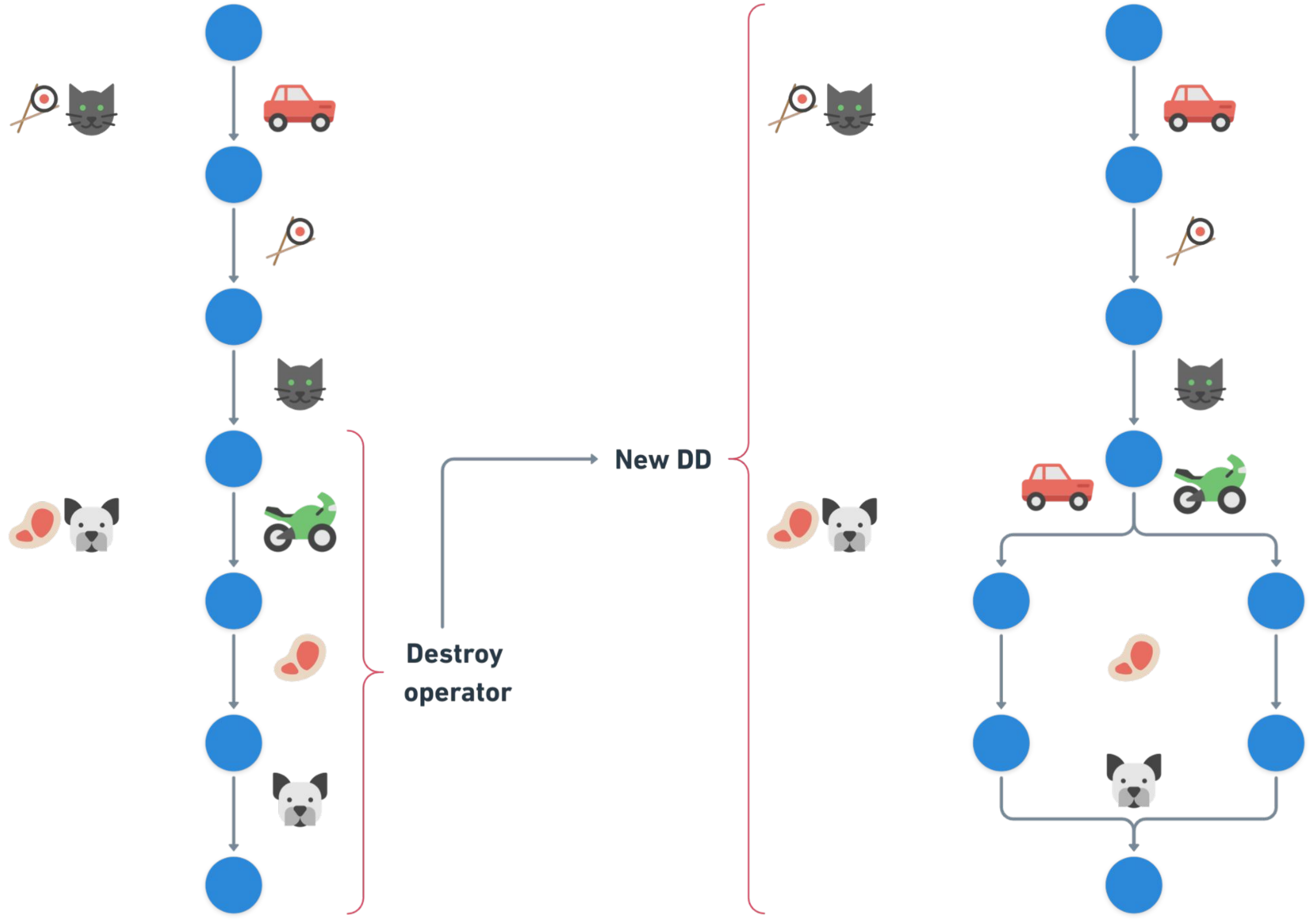


🆘 Help! I got stuck in a local optimum!

Can we combine metaheuristics with DDs?

- DD: exact solver, controls search, **functions as repair operator**
- ALNS: improves on incumbents, **only needs destroy operators**





New DD
Destroy operator

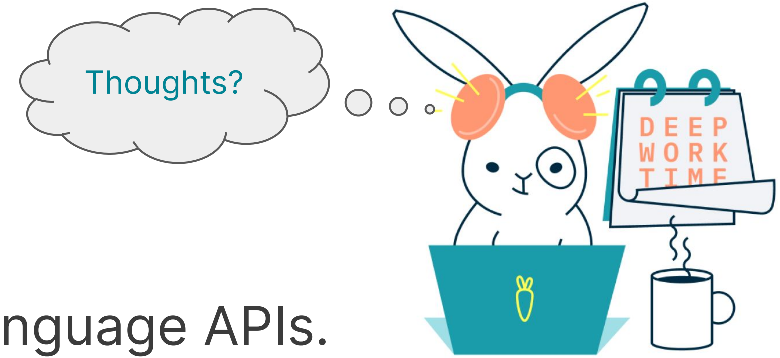
🧪 What went well?

- No data translation! Model directly on operational data.
- Good performance with lots of side constraints and flexibility
- DD + ALNS fit well together. It feels like neighborhood search can fit directly on the diagram.
- Layers and ordering are useful for inference.

💔 What needs work?

- Custom modeling. Statesplosion.
- Heuristics and reduction are too problem specific.
- Relaxation techniques hard to apply to arbitrary data.
- Immutable data isn't always great.

What’s next for Hop?



- Higher level modeling layer, language APIs.
- Tighten definitions of states and transitions. Provide custom data that doesn’t drive the search.
- Figure out how to combine automatic merging and state expansion.
- v1 end of year?™

Thank you!