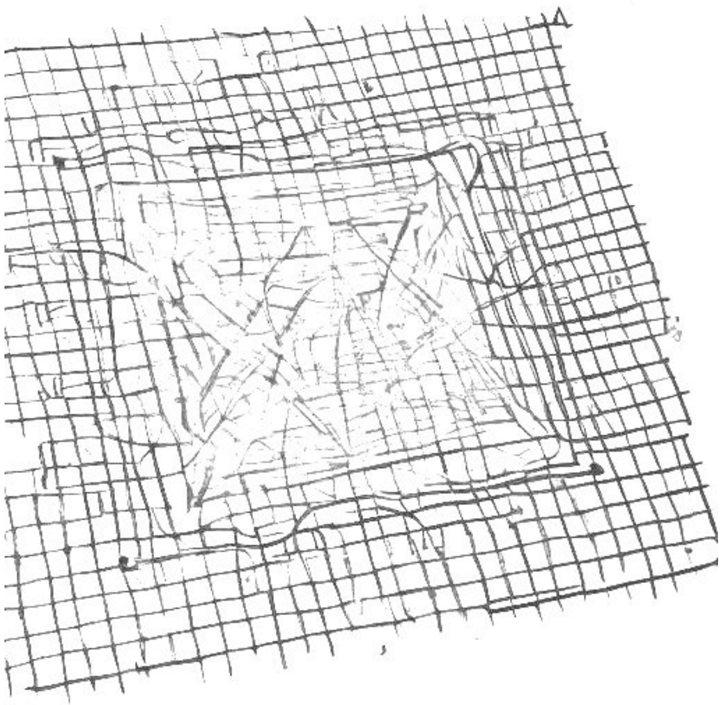
Developing

Neural Algorithmic Reasoning

Petar Veličković
Andreea Deac
Andrew Dudzik

Learning on Graphs Conference
10 December 2022



Motivation

What do we mean by *algorithm*?

Informally, an **algorithm** is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

**taken from Introduction to Algorithms, by Cormen, Leiserson, Rivest and Stein.*

What do we mean by *algorithm*?

Informally, an **algorithm** is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

An algorithm is thus a **sequence of computational steps** that transform the input into the output.

**taken from Introduction to Algorithms, by Cormen, Leiserson, Rivest and Stein.*

What do we mean by *algorithm*?

Informally, an **algorithm** is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

An algorithm is thus a **sequence of computational steps** that transform the input into the output.

An algorithm can be specified in English, as a computer program, or even as a hardware design. The only requirement is that the specification must provide a **precise** description of the computational procedure to be followed.

**taken from Introduction to Algorithms, by Cormen, Leiserson, Rivest and Stein.*

What do we mean by *algorithm*?

A common example of an algorithmic task is the **sorting** problem:

- **Input:** A sequence of n numbers $[a_1, a_2, \ldots, a_n]$
- **Output:** A permutation (reordering) $[a'_1, a'_2, \ldots, a'_n]$ of the input sequence, such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

**taken from Introduction to Algorithms, by Cormen, Leiserson, Rivest and Stein.*

What do we mean by *algorithm*?

A common example of an algorithmic task is the **sorting** problem:

- **Input:** A sequence of *n* numbers $[a_1, a_2, \ldots, a_n]$
- **Output:** A permutation (reordering) $[a'_1, a'_2, \ldots, a'_n]$ of the input sequence, such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

One algorithm that solves it is *insertion sort*.

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

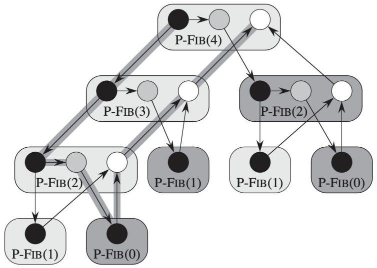
**taken from Introduction to Algorithms, by Cormen, Leiserson, Rivest and Stein.*

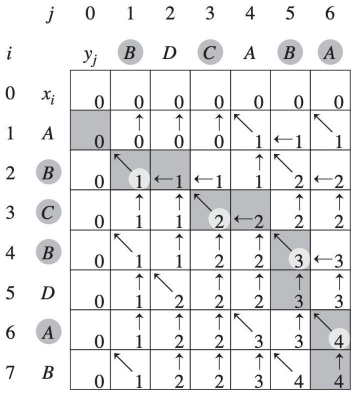
Why algorithms?

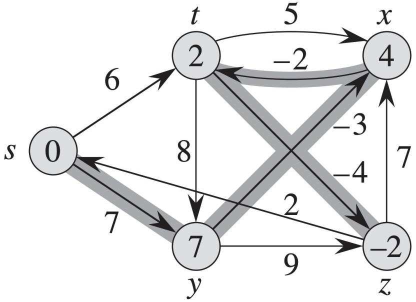
Essential “pure” forms of combinatorial reasoning

- ‘Timeless’ principles that remain, regardless of the model of computation
- Completely decoupled from any form of perception*

**though perception itself may also be expressed in the language of algorithms*







Why algorithms?

Essential “pure” forms of combinatorial reasoning

- ‘Timeless’ principles that remain, regardless of the model of computation
- Completely decoupled from any form of perception

Favourable properties

- Trivial **strong** generalisation
- **Compositionality** via subroutines
- Provable **correctness** and **performance** guarantees
- Interpretable **operations** / *pseudocode*

Why algorithms?

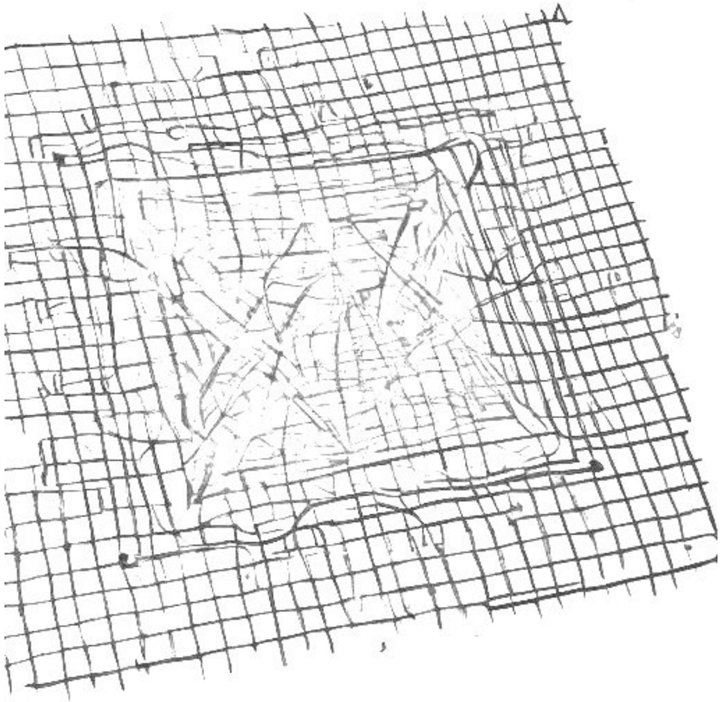
Essential “pure” forms of combinatorial reasoning

- ‘Timeless’ principles that remain, regardless of the model of computation
- Completely decoupled from any form of perception

Favourable properties

- Trivial **strong** generalisation
- **Compositionality** via subroutines
- Provable **correctness** and **performance** guarantees
- Interpretable **operations** / *pseudocode*

Hits close to home, for many of us :)



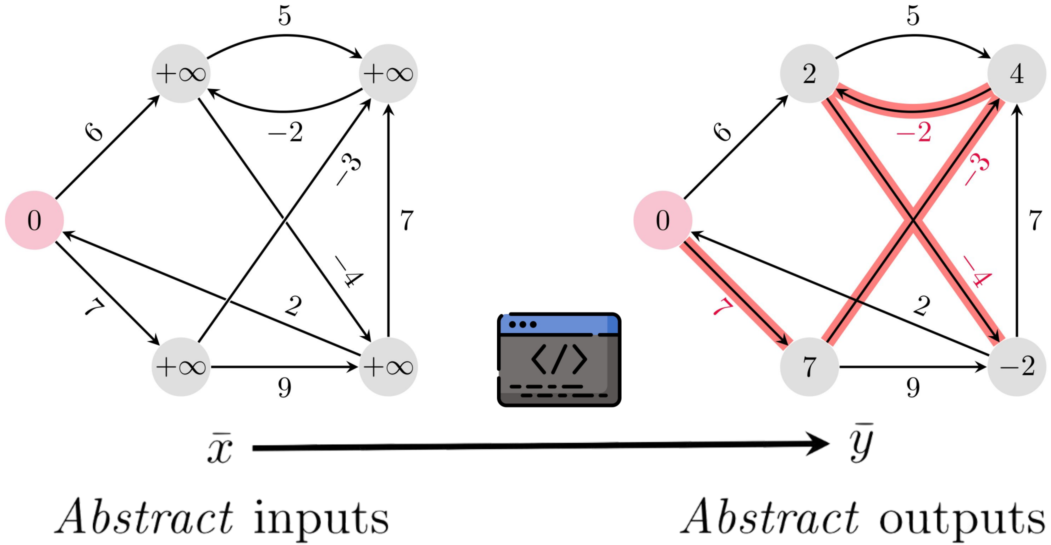
When do algorithms exhibit *flaws*?

A simple example

"Find the **optimal** path from A to B"

A simple example

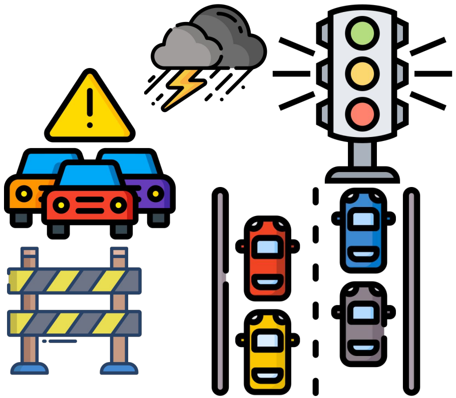
"Find the **optimal** path from A to B"



The theoretical computer scientist diligently uses the Dijkstra hammer!

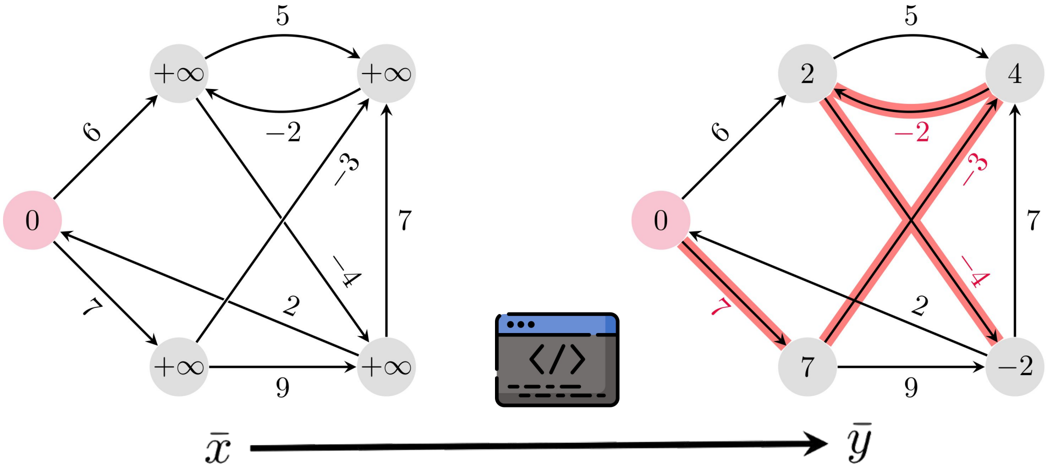
A simple example

"Find the **optimal** path from A to B"



x

Natural inputs



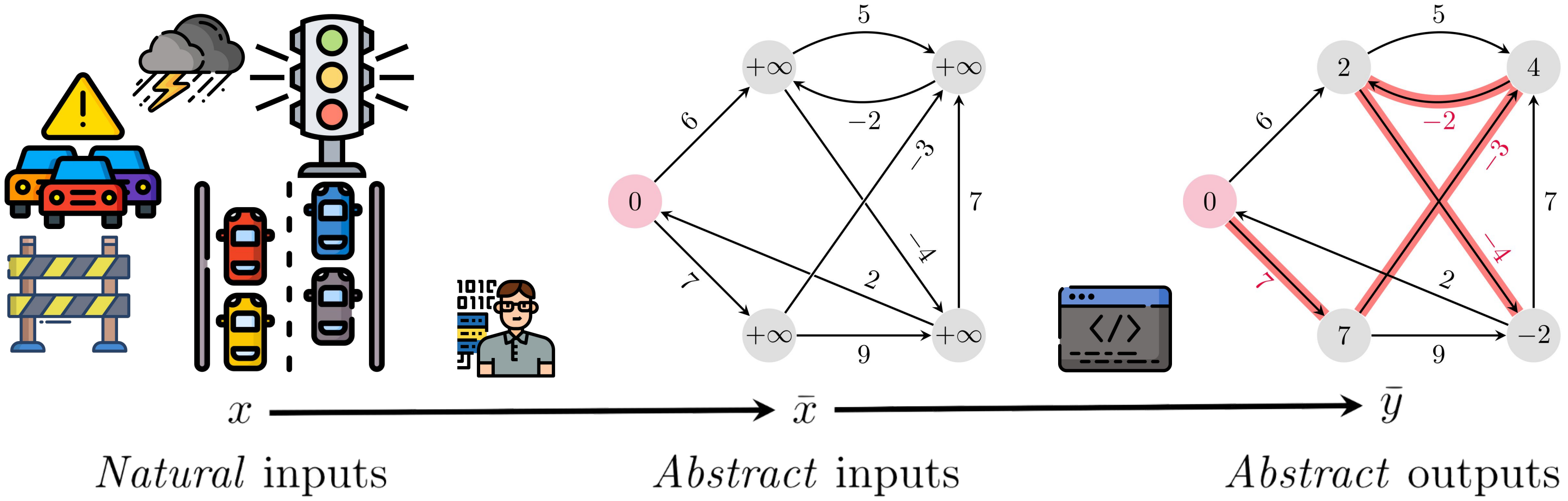
Abstract inputs

Abstract outputs

*This kind of question usually hides the **real-world** problem underneath…*

A simple example

“Find the **optimal** path from A to B”



*Can we ever hope to **manually** do the mapping necessary?*

Not really... (known at least since *1955*)

SECRET

U. S. AIR FORCE
PROJECT RAND
RESEARCH MEMORANDUM

FUNDAMENTALS OF A METHOD FOR EVALUATING
RAIL NET CAPACITIES (U)

T. E. Harris
F. S. Ross

RM-1573

October 24, 1955 Copy No. 37

This material contains information affecting the national defense of the United States within the meaning of the espionage laws, Title 18 U.S.C., Secs 793 and 794, the transmission or the revelation of which in any manner to an unauthorized person is prohibited by law

II. THE ESTIMATING OF RAILWAY CAPACITIES

The evaluation of both railway system and individual track capacities is, to a considerable extent, an art. The authors know of no tested mathematical model or formula that includes all of the variations and imponderables that must be weighed.* Even when the individual has been closely associated with the particular territory he is evaluating, the final answer, however accurate, is largely one of judgment and experience.

The core problem

A **divide** between algorithms and real-world tasks they were *designed* to solve!

Satisfying the algorithm's *strict* preconditions may drastically lose information.

The core problem

A **divide** between algorithms and real-world tasks they were *designed* to solve!

Satisfying the algorithm's *strict* preconditions may drastically lose information.

It doesn't matter that the algorithm is provably correct,
if it's executed on the <u>wrong</u> inputs!

The core problem

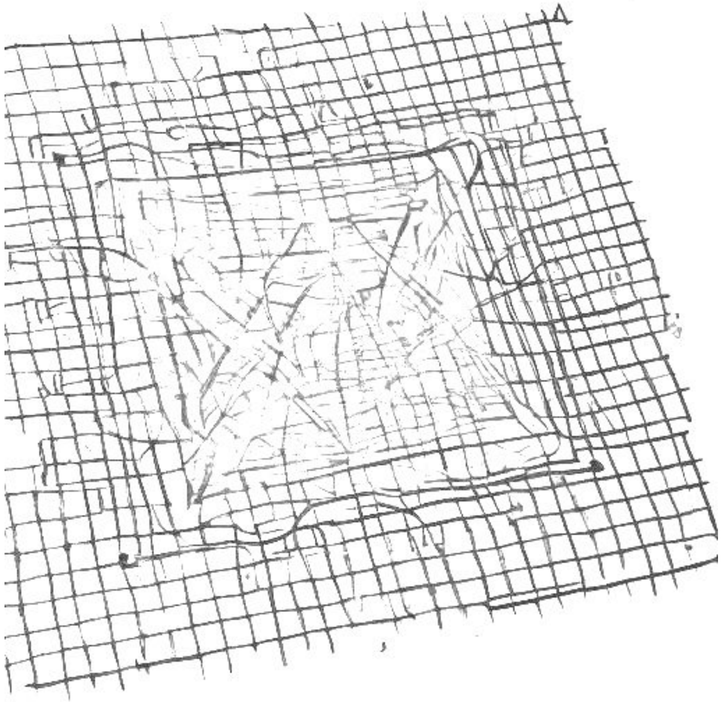
A **divide** between algorithms and real-world tasks they were *designed* to solve!

Satisfying the algorithm's *strict* preconditions may drastically lose information.

It doesn't matter that the algorithm is provably correct,
if it's executed on the <u>wrong</u> inputs!

This is tricky even without considering issues like *partially observable* data, etc.

In this tutorial, we will attack this core problem by **neuralising** the algorithm

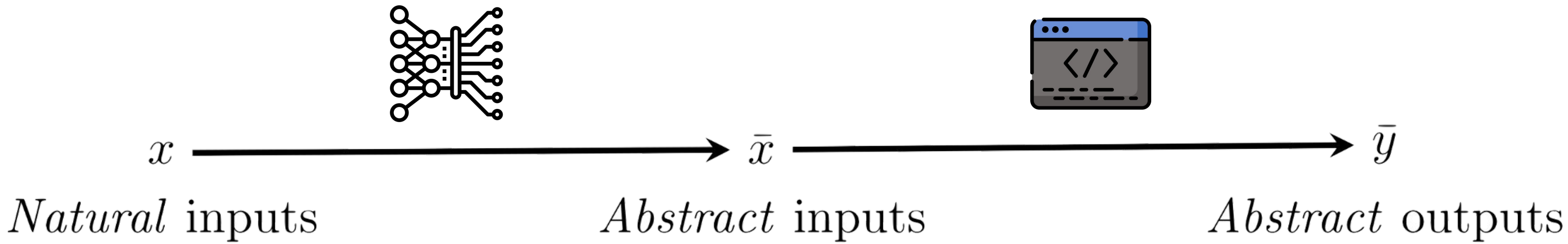


Neuralising an algorithm

Attacking the core problem

The problem rests on **manual** *feature engineering* of **raw** data. This is what neural networks were designed to solve! :)

Let’s replace our feature extractor with a **neural network**.

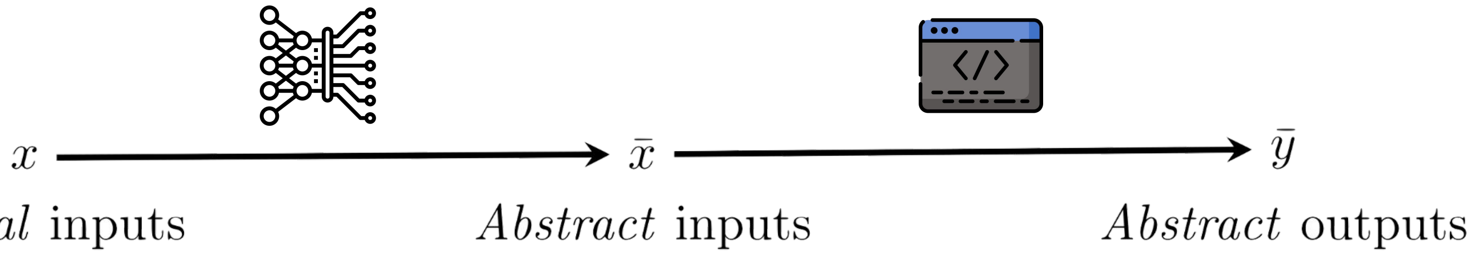


Train the neural network using gradient descent.

Attacking the core problem

The problem rests on **manual** *feature engineering* of **raw** data. This is what neural networks were designed to solve! :)

Let's replace our feature extractor with a **neural network**.

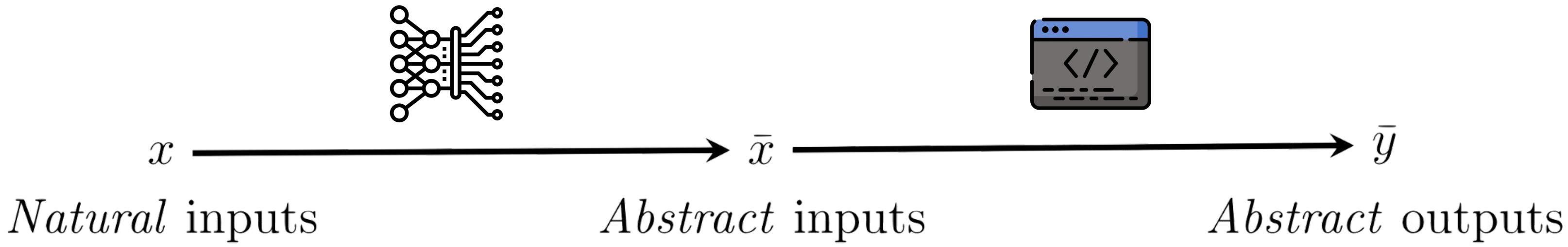


This used to be problematic due to *discreteness* of the algorithm.
Nowadays, there exist established ways to **backpropagate** through arbitrary black-box optimisation functions (see, e.g., Vlastelica *et al.*, ICLR'20).

The *algorithmic bottleneck* (informally)

Fundamental issue: our pipeline strongly *commits* to using the algorithm.

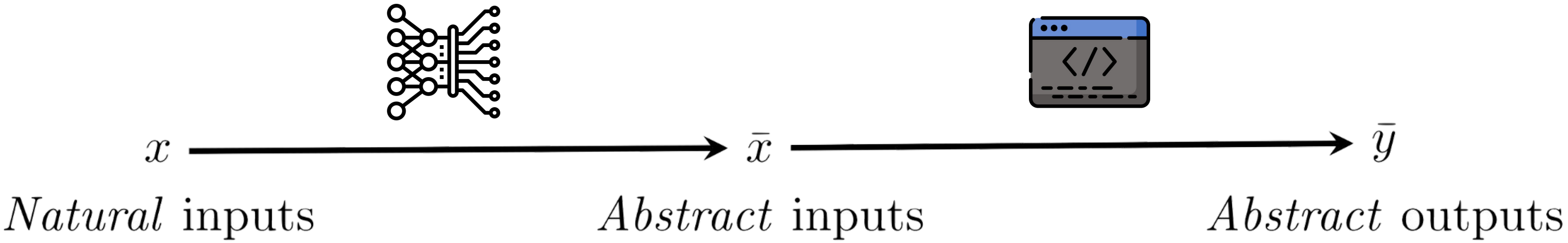
Once we compute the inputs to the algorithm, we are fully trusting what comes out of it, with no way to revert any mistakes!



The *algorithmic bottleneck* (informally)

Fundamental issue: our pipeline strongly *commits* to using the algorithm.

Once we compute the inputs to the algorithm, we are fully trusting what comes out of it, with no way to revert any mistakes!

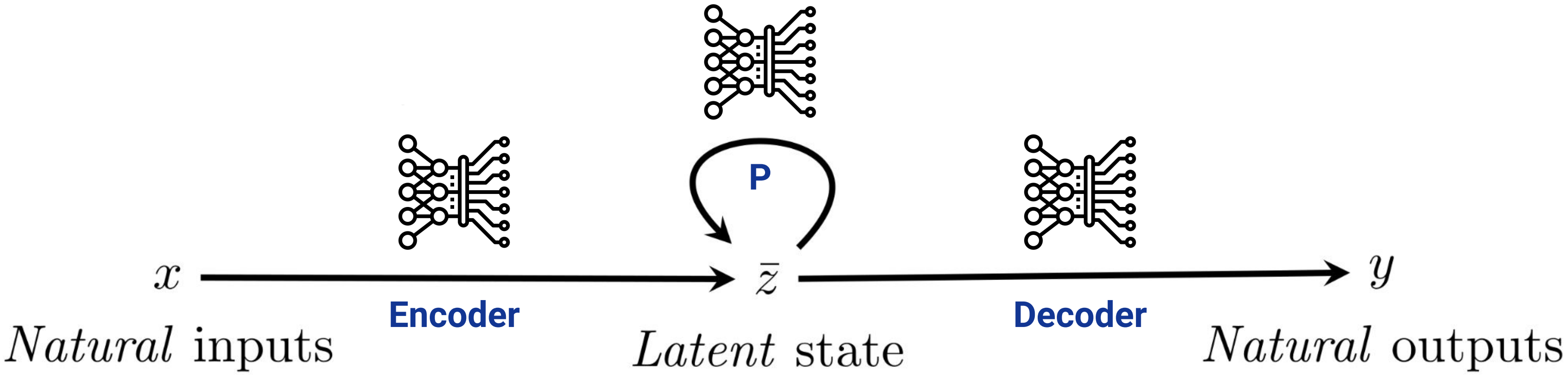


In many scenarios, this can lead to the **algorithmic bottleneck** problem.
What if there is *insufficient training data* to properly estimate the inputs?
What if we need to run *more than one* algorithm?

Breaking the bottleneck

Neural networks derive flexibility from their **high-dimensional** latents, $z \in \mathbb{R}^m$.

If any component of the latent is poorly predicted, others can step in!

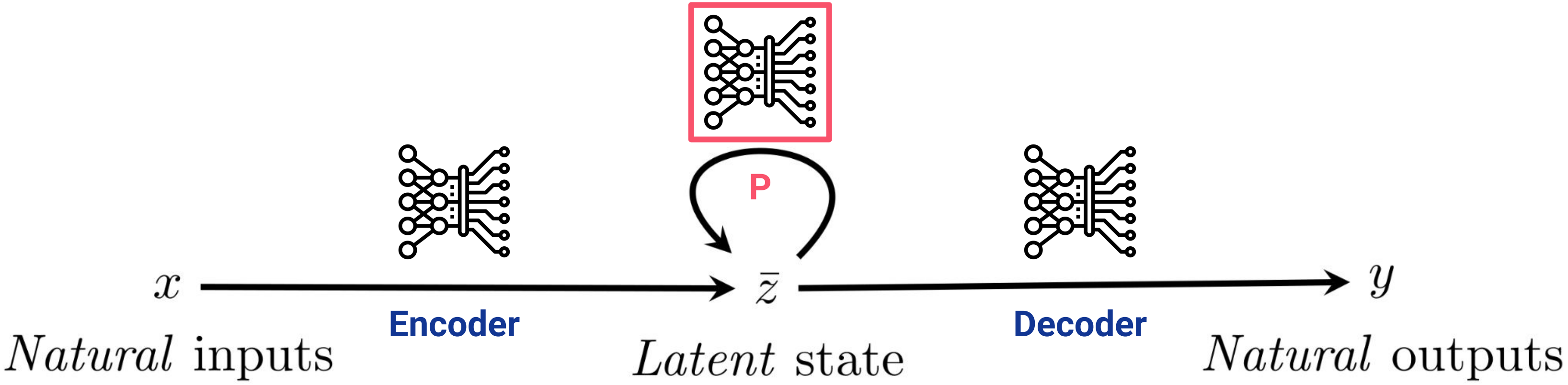


To break the bottleneck, replace the algorithm with a **processor network, P**.

(The setting naturally aligns with the encode-process-decode paradigm (Hamrick et al., CSS'18))

Breaking the bottleneck

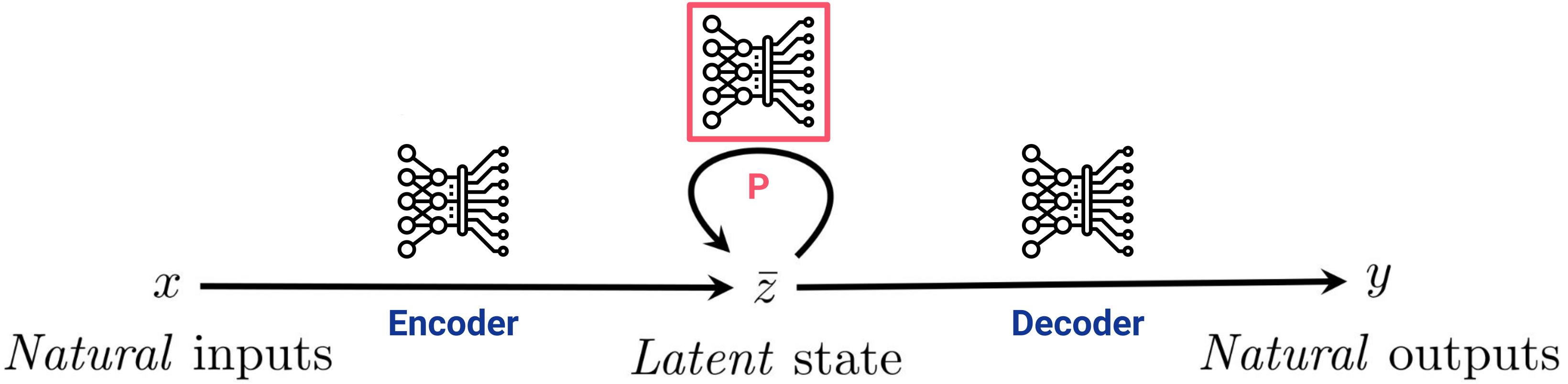
Assuming we can obtain a processor, **P** : $\mathbb{R}^m \rightarrow \mathbb{R}^m$, such that it somehow *aligns* with the algorithmic steps, we have everything we need!



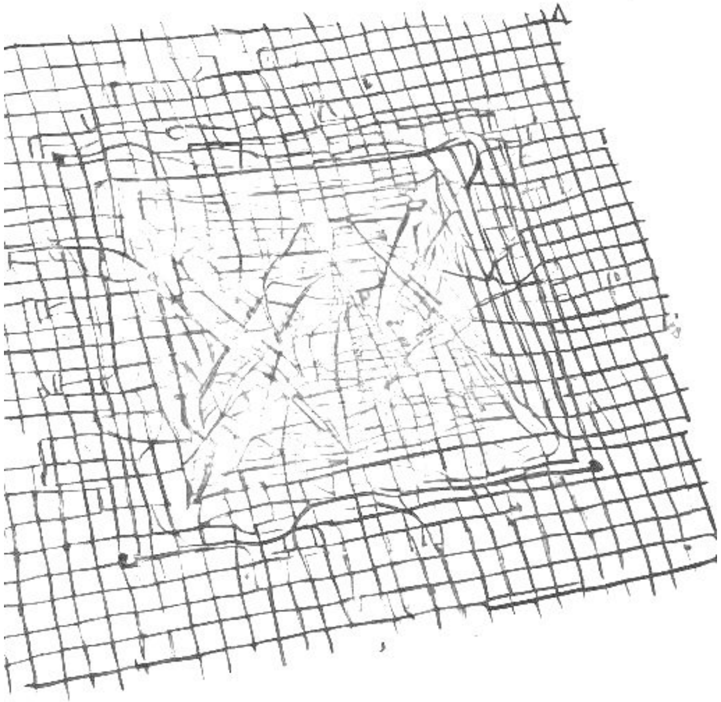
(differentiable, no bottlenecks, can fit residual algorithms by skip-connecting ***P****)*

Breaking the bottleneck

Assuming we can obtain a processor, **P** : $\mathbb{R}^m \rightarrow \mathbb{R}^m$, such that it somehow *aligns* with the algorithmic steps, we have everything we need!



How to obtain **latent-state neural networks** that **align** with *algorithms*?



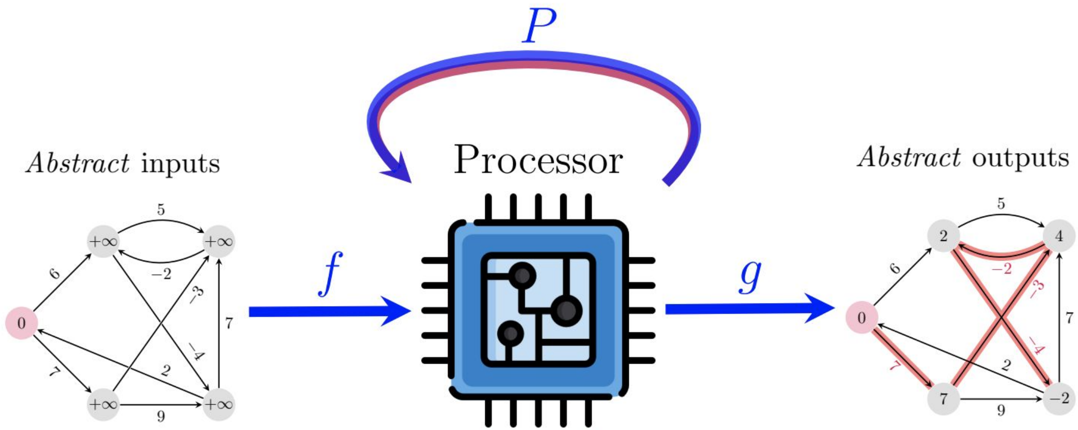
Neural Algorithmic Reasoning

Why do we need a new field?

What is different about learning a good **P**, compared to any other ML task?

It needs to imitate the steps of the target algorithm *faithfully*—which means it must **extrapolate** well beyond the training set!

This is a regime in which neural nets tend to **struggle**!



Why do we need a new field?

What is different about learning a good **P**, compared to any other ML task?

It needs to imitate the steps of the target algorithm *faithfully*—which means it must **extrapolate** well beyond the training set!

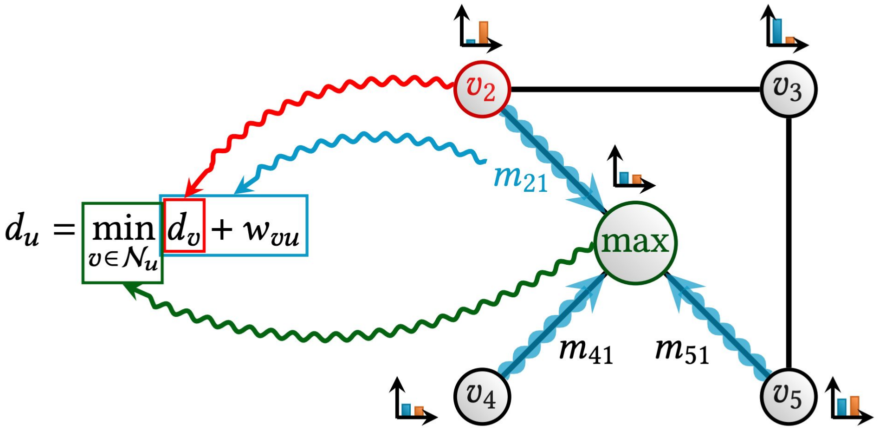
This is a regime in which neural nets tend to **struggle**!

Neural Algorithmic Reasoning is an emerging area that attempts to build potent processor networks **P**. This can be done in a variety of ways:

- Architecture choice of **P**, encoder or decoder
- Choice of input features / their transformations
- Training schedule for the overall system

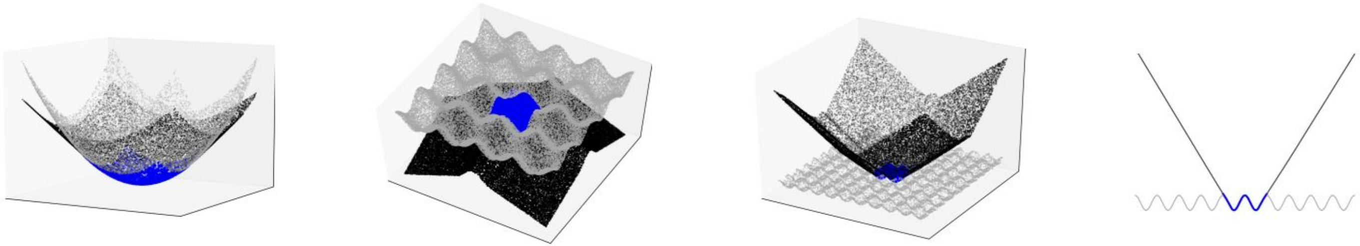
What do we know, theoretically?

- **Algorithmic alignment**
  - Better structural alignment of the model to the algorithm **implies** better generalisation
  - Informal observation: GNNs align well with *dynamic programming*!
  - Xu *et al.*, “What Can Neural Networks Reason About?”. ICLR’20 *[See also: Part III of tutorial.]*



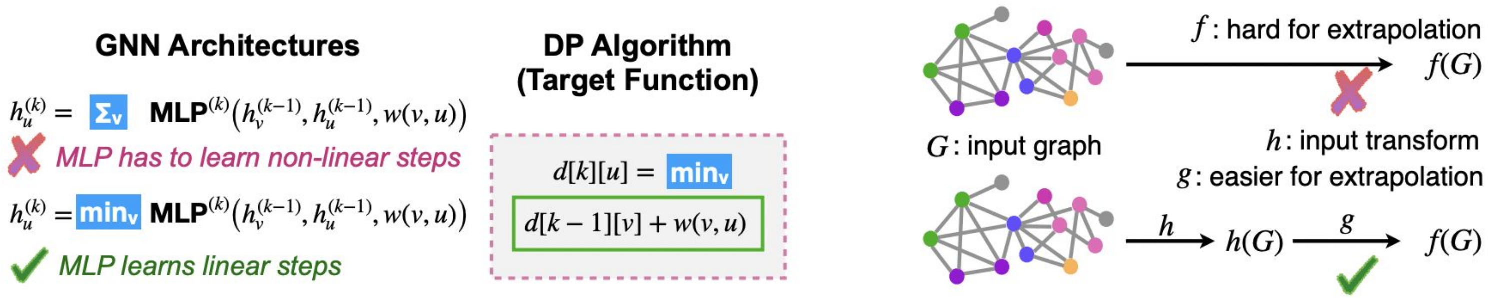
What do we know, theoretically?

- **Algorithmic alignment**
  - Better structural alignment of the model to the algorithm **implies** better generalisation
  - Informal observation: GNNs align well with *dynamic programming*!
  - Xu *et al.*, “What Can Neural Networks Reason About?”. ICLR’20 *[See also: Part III of tutorial.]*
- **Linear algorithmic alignment**
  - To **extrapolate**, the target functions for parts of our (G)NN must be **linear** (for ReLU MLPs).
  - Xu *et al.*, “How Neural Networks Extrapolate…”. ICLR’21



What do we know, theoretically?

- **Algorithmic alignment**
  - Better structural alignment of the model to the algorithm **implies** better generalisation
  - Informal observation: GNNs align well with *dynamic programming*!
  - Xu *et al.*, "What Can Neural Networks Reason About?". ICLR'20 *[See also: Part III of tutorial.]*
- **Linear algorithmic alignment**
  - To **extrapolate**, the target functions for parts of our (G)NN must be **linear** (for ReLU MLPs).
  - Xu *et al.*, "How Neural Networks Extrapolate…". ICLR'21

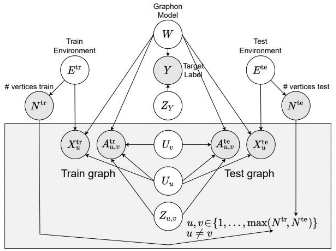


(a) Network architecture

(b) Input representation

What do we know, theoretically?

- **Algorithmic alignment**
  - Better structural alignment of the model to the algorithm **implies** better generalisation
  - Informal observation: GNNs align well with *dynamic programming*!
  - Xu *et al.*, "What Can Neural Networks Reason About?". ICLR'20 *[See also: Part III of tutorial.]*
- **Linear algorithmic alignment**
  - To **extrapolate**, the target functions for parts of our (G)NN must be **linear** (for ReLU MLPs).
  - Xu *et al.*, "How Neural Networks Extrapolate…". ICLR'21
- **Causality-based alignment**
  - In general, to extrapolate, we would need to carry a **causal model** of distribution shift
  - Bevilacqua, Zhou and Ribeiro, "Size-invariant Graph Representations…". ICML'21



What do we know, theoretically?

- **Algorithmic alignment**
  - Better structural alignment of the model to the algorithm **implies** better generalisation
  - Informal observation: GNNs align well with *dynamic programming*!
  - Xu *et al.*, “What Can Neural Networks Reason About?”. ICLR’20 *[See also: Part III of tutorial.]*
- **Linear algorithmic alignment**
  - To **extrapolate**, the target functions for parts of our (G)NN must be **linear** (for ReLU MLPs).
  - Xu *et al.*, “How Neural Networks Extrapolate…”. ICLR’21
- **Causality-based alignment**
  - In general, to extrapolate, we would need to carry a **causal model** of distribution shift
  - Bevilacqua, Zhou and Ribeiro, “Size-invariant Graph Representations…”. ICML’21
- **Permutation compatibility**
  - We usually assume that the GNN is appropriately featurised when executing the algorithm.
  - If a task is **permutation-compatible**, then the choice of features is not even relevant!
  - Fereydounian *et al.*, “What Functions Can Graph Neural Networks Generate?”. 2022

What do we know, empirically?

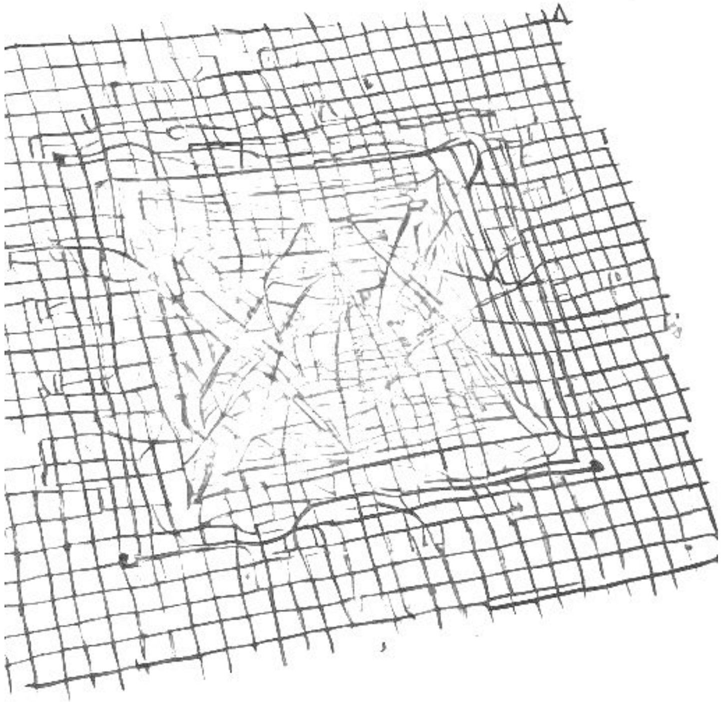
- **Better-aligned architectures indeed yield better processors!**
    - Neural Shuffle-Exchange Networks (Freivalds *et al.*, NeurIPS’19) — *Linearithmic algorithms*
    - Neural Execution of Graph Algorithms (Veličković *et al.*, ICLR’20) — *Dynamic programming*
    - PrediNet (Shanahan *et al.*, ICML’20) — *Predicate logic*
    - IterGNNs (Tang *et al.*, NeurIPS’20) — *Iterative algorithms*
    - Pointer Graph Networks (Veličković *et al.*, NeurIPS’20) — *Pointer-based data structures*
    - Persistent Message Passing (Strathmann *et al.*, ICLR’21 SimDL) — *Persistent data structures*

What do we know, empirically?

- **Better-aligned architectures indeed yield better processors!**
  - Neural Shuffle-Exchange Networks (Freivalds *et al.*, NeurIPS'19)
  - Neural Execution of Graph Algorithms (Veličković *et al.*, ICLR'20)
  - PrediNet (Shanahan *et al.*, ICML'20)
  - IterGNNs (Tang *et al.*, NeurIPS'20)
  - Pointer Graph Networks (Veličković *et al.*, NeurIPS'20)
  - Persistent Message Passing (Strathmann *et al.*, ICLR'21 SimDL)
- **Careful modifications to the training regime can yield better processors!**
  - Unsupervised learning (Karalias and Loukas, NeurIPS'20)
  - Self-supervised learning (Yehudai *et al.*, ICML'21)
  - Shift-size regularisation (Buffelli *et al.*, NeurIPS'22)
  - Recall (Bansal, Schwarzschild *et al.*, NeurIPS'22)

What do we know, empirically?

- **Better-aligned architectures indeed yield better processors!**
  - Neural Shuffle-Exchange Networks (Freivalds *et al.*, NeurIPS’19)
  - Neural Execution of Graph Algorithms (Veličković *et al.*, ICLR’20)
  - PrediNet (Shanahan *et al.*, ICML’20)
  - IterGNNs (Tang *et al.*, NeurIPS’20)
  - Pointer Graph Networks (Veličković *et al.*, NeurIPS’20)
  - Persistent Message Passing (Strathmann *et al.*, ICLR’21 SimDL)
- **Careful modifications to the training regime can yield better processors!**
  - Unsupervised learning (Karalias and Loukas, NeurIPS’20)
  - Self-supervised learning (Yehudai *et al.*, ICML’21)
  - Shift-size regularisation (Buffelli *et al.*, NeurIPS’22)
  - Recall (Bansal, Schwarzschild *et al.*, NeurIPS’22)
- **We can also learn *multiple* algorithms at once!**
  - NeuralExecutor++ (Xhonneux *et al.*, NeurIPS’21)
  - A Generalist Neural Algorithmic Learner (Ibarz *et al.*, LoG’22)



The CLRS-30 Benchmark

Benchmarking algorithmic reasoners

Sorting: Insertion sort, bubble sort, heapsort (Williams, 1964), quicksort (Hoare, 1962).

Searching: Minimum, binary search, quickselect (Hoare, 1961).

Divide and Conquer (D&C): Maximum subarray (Kadane's variant (Bentley, 1984)).

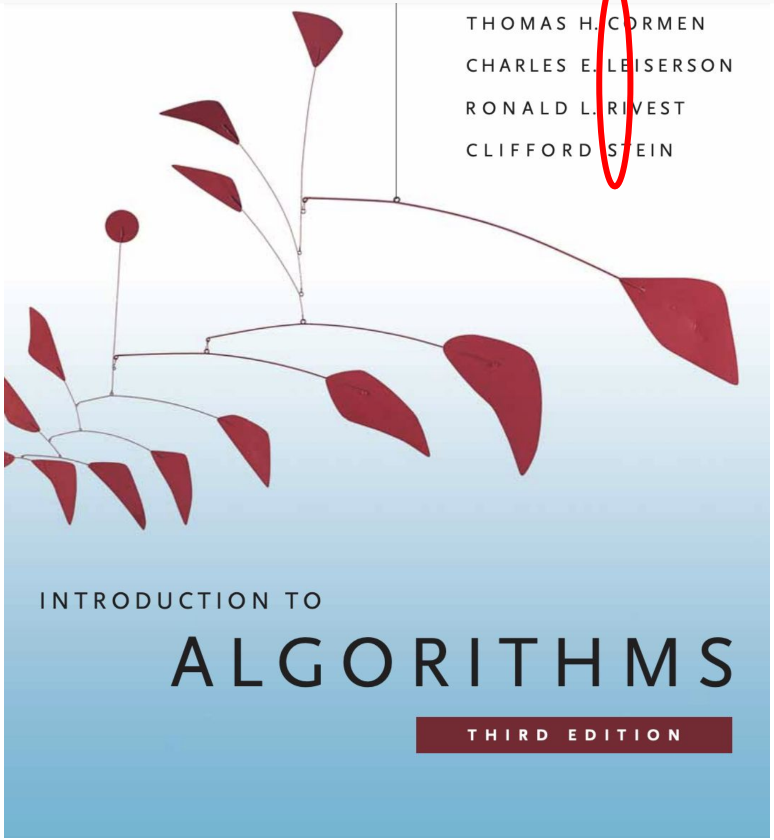
Greedy: Activity selection (Gavril, 1972), task scheduling (Lawler, 1985).

Dynamic Programming: Matrix chain multiplication, longest common subsequence, optimal binary search tree (Aho et al., 1974).

Graphs: Depth-first and breadth-first search (Moore, 1959), topological sorting (Knuth, 1973), articulation points, bridges, Kosaraju's strongly-connected components algorithm (Aho et al., 1974), Kruskal's and Prim's algorithms for minimum spanning trees (Kruskal, 1956; Prim, 1957), Bellman-Ford and Dijkstra's algorithms for single-source shortest paths (Bellman, 1958; Dijkstra et al., 1959) (+ directed acyclic graphs version), Floyd-Warshall algorithm for all-pairs shortest paths (Floyd, 1962).

Strings: Naïve string matching, Knuth-Morris-Pratt (KMP) string matcher (Knuth et al., 1977).

Geometry: Segment intersection, Convex hull algorithms: Graham scan (Graham, 1972), Jarvis' march (Jarvis, 1973).



Benchmarking algorithmic reasoners

https://github.com/deepmind/clrs

The CLRS Algorithmic Reasoning Benchmark

Petar Veličković [1] Adrià Puigdomènech Badia [1] David Budden [1]
Razvan Pascanu [1] Andrea Banino [1] Misha Dashevskiy [1] Raia Hadsell [1] Charles Blundell [1]

Representation

- All algorithms have been boiled down to a common **graph representation**

Representation

- All algorithms have been boiled down to a common **graph representation**
- Each algorithm is specified by a fixed number of “probes”.
  - A probe is a specific variable that is tracked during the algorithm’s execution.
  - The model may be asked to use those variables as input, predict them as output, or both.
- Specifying the task’s probes **uniquely** determines the dataset shape for this task, the model’s encoder/decoder architectures, and loss functions!
  - We can think of CLRS-30 as a “dataset / baseline generator” rather than a (single) dataset!

Representation

- All algorithms have been boiled down to a common **graph representation**
- For example, the spec of insertion sort consists of the following 6 probes:

```
'pos': (Stage.INPUT, Location.NODE, Type.SCALAR)    -> the id of each node
```

Representation

- All algorithms have been boiled down to a common **graph representation**
- For example, the spec of insertion sort consists of the following 6 probes:

```
'pos': (Stage.INPUT, Location.NODE, Type.SCALAR)    -> the id of each node
'key': (Stage.INPUT, Location.NODE, Type.SCALAR)    -> the values to sort
```

Representation

- All algorithms have been boiled down to a common **graph representation**
- For example, the spec of insertion sort consists of the following 6 probes:

```
'pos': (Stage.INPUT, Location.NODE, Type.SCALAR)     -> the id of each node
'key': (Stage.INPUT, Location.NODE, Type.SCALAR)     -> the values to sort
'pred': (Stage.OUTPUT, Location.NODE, Type.POINTER) -> the final node order
```

Representation

- All algorithms have been boiled down to a common **graph representation**
- For example, the spec of insertion sort consists of the following 6 probes:

```
'pos': (Stage.INPUT, Location.NODE, Type.SCALAR)     -> the id of each node
'key': (Stage.INPUT, Location.NODE, Type.SCALAR)     -> the values to sort
'pred': (Stage.OUTPUT, Location.NODE, Type.POINTER) -> the final node order
'pred_h': (Stage.HINT, Location.NODE, Type.POINTER) -> the node order along execution
```

Representation

- All algorithms have been boiled down to a common **graph representation**
- For example, the spec of insertion sort consists of the following 6 probes:

```
'pos': (Stage.INPUT, Location.NODE, Type.SCALAR)      -> the id of each node
'key': (Stage.INPUT, Location.NODE, Type.SCALAR)      -> the values to sort
'pred': (Stage.OUTPUT, Location.NODE, Type.POINTER)   -> the final node order
'pred_h': (Stage.HINT, Location.NODE, Type.POINTER)   -> the node order along execution
'i': (Stage.HINT, Location.NODE, Type.MASK_ONE)       -> index for insertion
```

Representation

- All algorithms have been boiled down to a common **graph representation**
- For example, the spec of insertion sort consists of the following 6 probes:

```
'pos': (Stage.INPUT, Location.NODE, Type.SCALAR)     -> the id of each node
'key': (Stage.INPUT, Location.NODE, Type.SCALAR)     -> the values to sort
'pred': (Stage.OUTPUT, Location.NODE, Type.POINTER) -> the final node order
'pred_h': (Stage.HINT, Location.NODE, Type.POINTER) -> the node order along execution
'i': (Stage.HINT, Location.NODE, Type.MASK_ONE)      -> index for insertion
'j': (Stage.HINT, Location.NODE, Type.MASK_ONE)      -> index tracking “sorted up to”
```

Representation

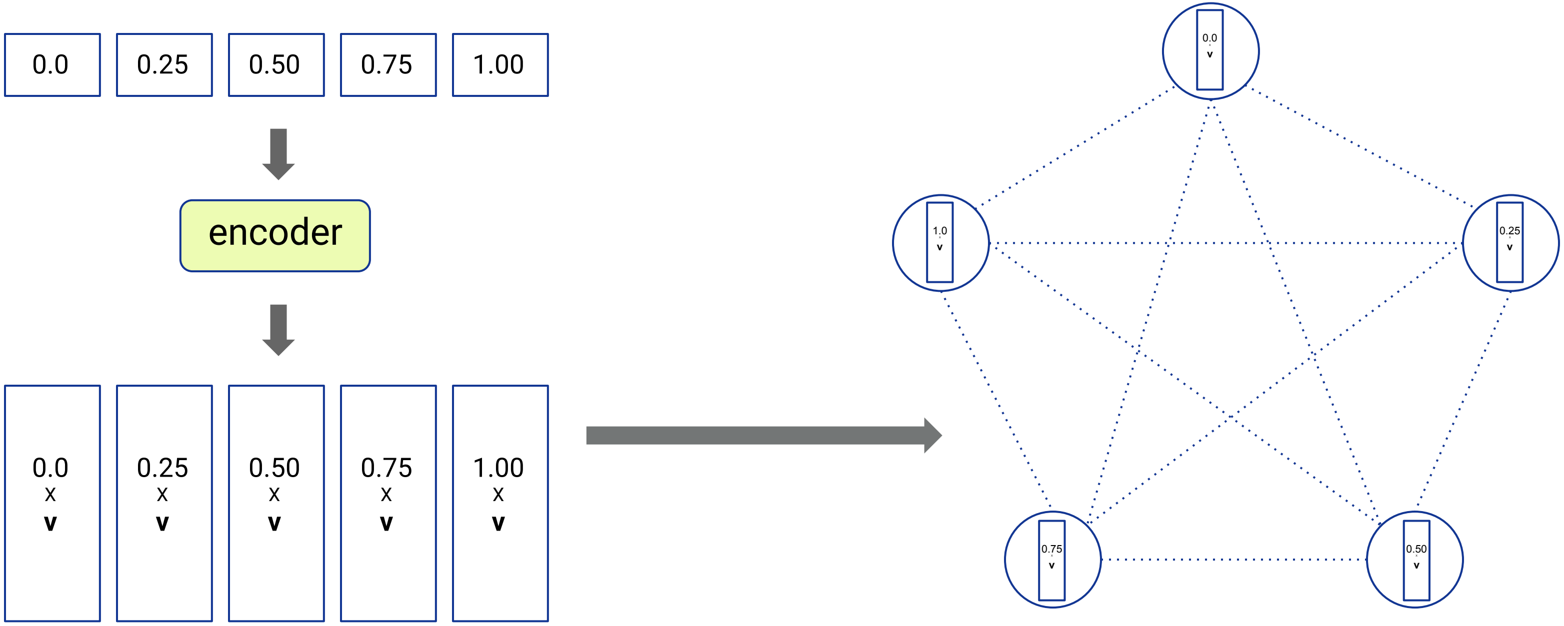
- All algorithms have been boiled down to a common **graph representation**
- For example, the spec of insertion sort consists of the following 6 probes:

```
'pos': (Stage.INPUT, Location.NODE, Type.SCALAR)     -> the id of each node
'key': (Stage.INPUT, Location.NODE, Type.SCALAR)     -> the values to sort
'pred': (Stage.OUTPUT, Location.NODE, Type.POINTER) -> the final node order
'pred_h': (Stage.HINT, Location.NODE, Type.POINTER) -> the node order along execution
'i': (Stage.HINT, Location.NODE, Type.MASK_ONE)      -> index for insertion
'j': (Stage.HINT, Location.NODE, Type.MASK_ONE)      -> index tracking “sorted up to”
```

- A probe can be **input**, **output** or **hint**. Inputs and outputs are fixed during algorithm execution, the hints change during execution - they specify the algorithm (e.g., sorting algorithms differ only in their hints).

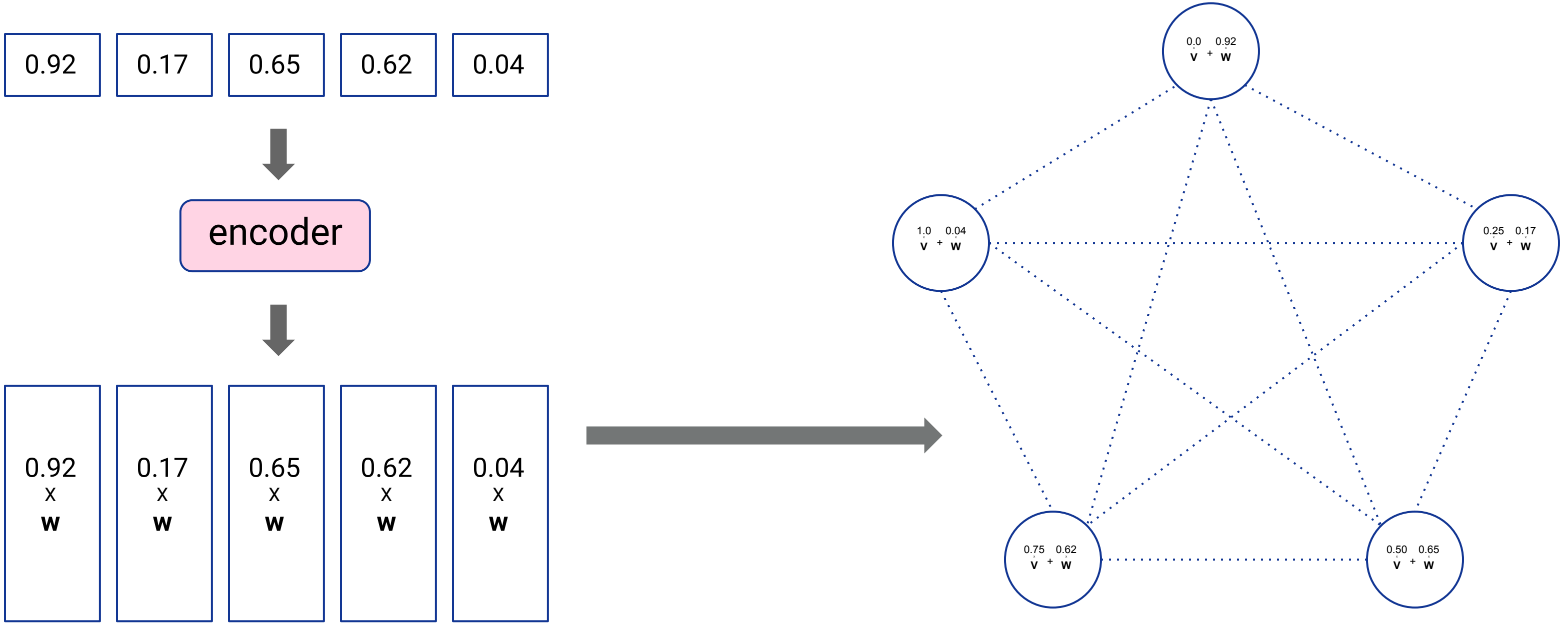
Representation: *encoding*

```
'pos': (Stage.INPUT, Location.NODE, Type.SCALAR)
```



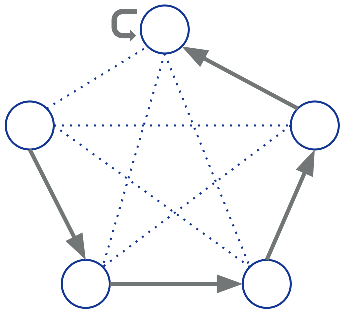
Representation: *encoding*

```
'pos': (Stage.INPUT, Location.NODE, Type.SCALAR)
'key': (Stage.INPUT, Location.NODE, Type.SCALAR)
```



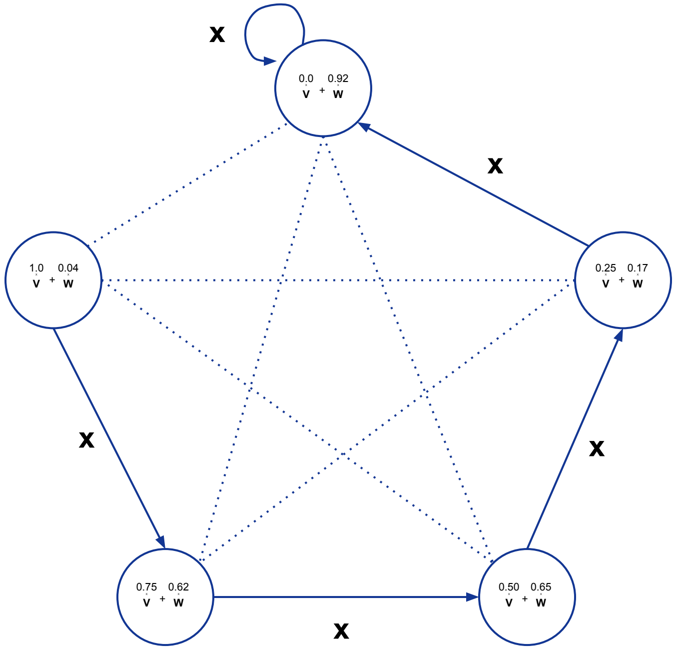
Representation: *encoding*

```
'pos': (Stage.INPUT, Location.NODE, Type.SCALAR)
'key': (Stage.INPUT, Location.NODE, Type.SCALAR)
'pred_h': (Stage.HINT, Location.NODE, Type.POINTER)
```



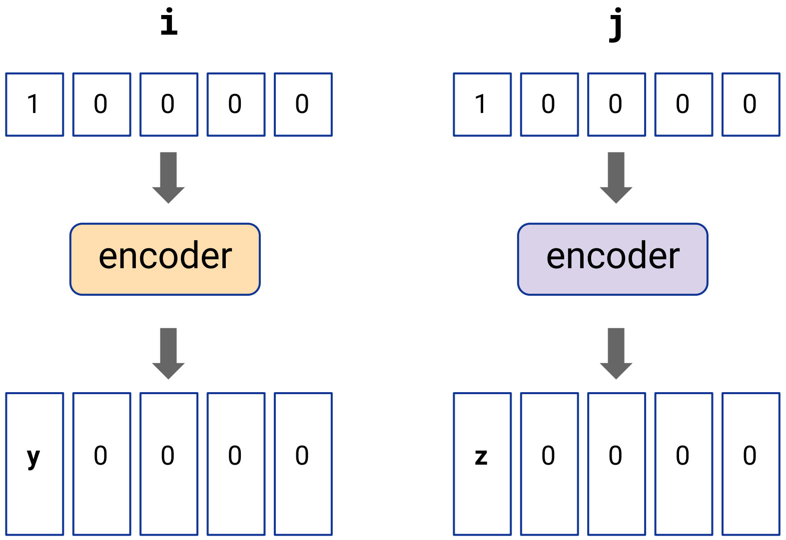
$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

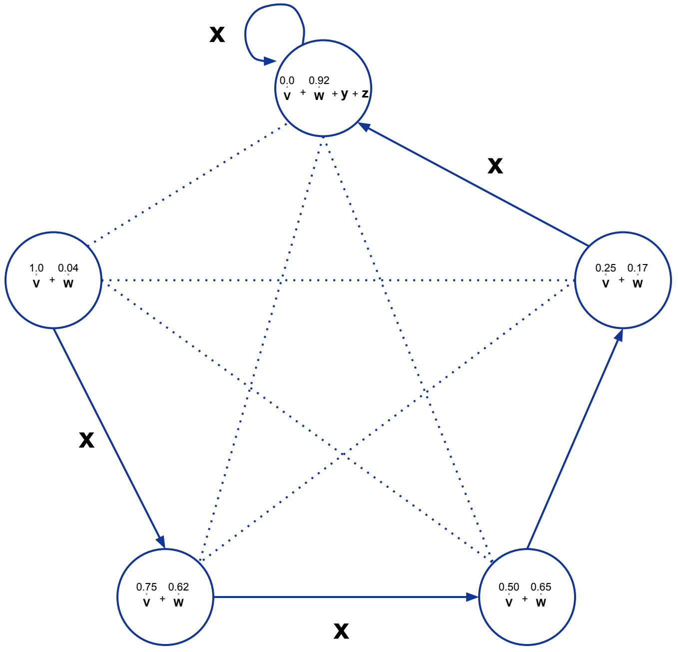
encoder



Representation: *encoding*

```
'pos': (Stage.INPUT, Location.NODE, Type.SCALAR)
'key': (Stage.INPUT, Location.NODE, Type.SCALAR)
'pred_h': (Stage.HINT, Location.NODE, Type.POINTER)
'i': (Stage.HINT, Location.NODE, Type.MASK_ONE)
'j': (Stage.HINT, Location.NODE, Type.MASK_ONE)
```

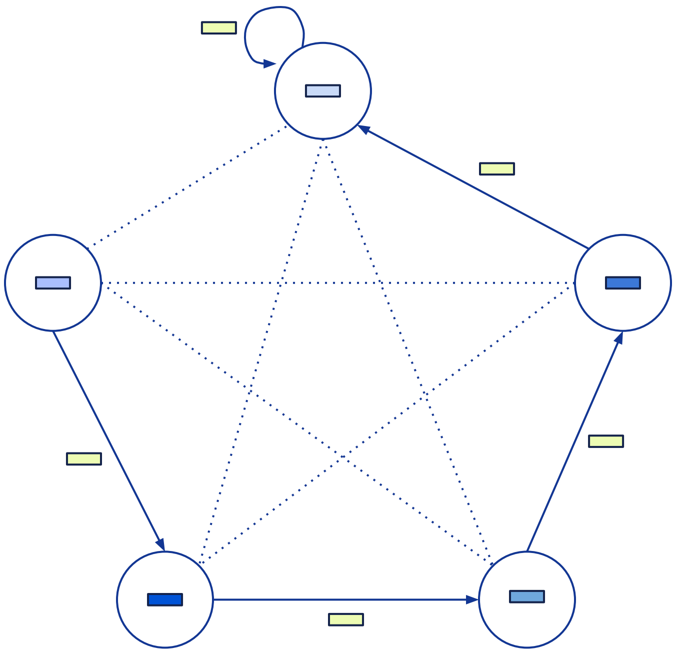




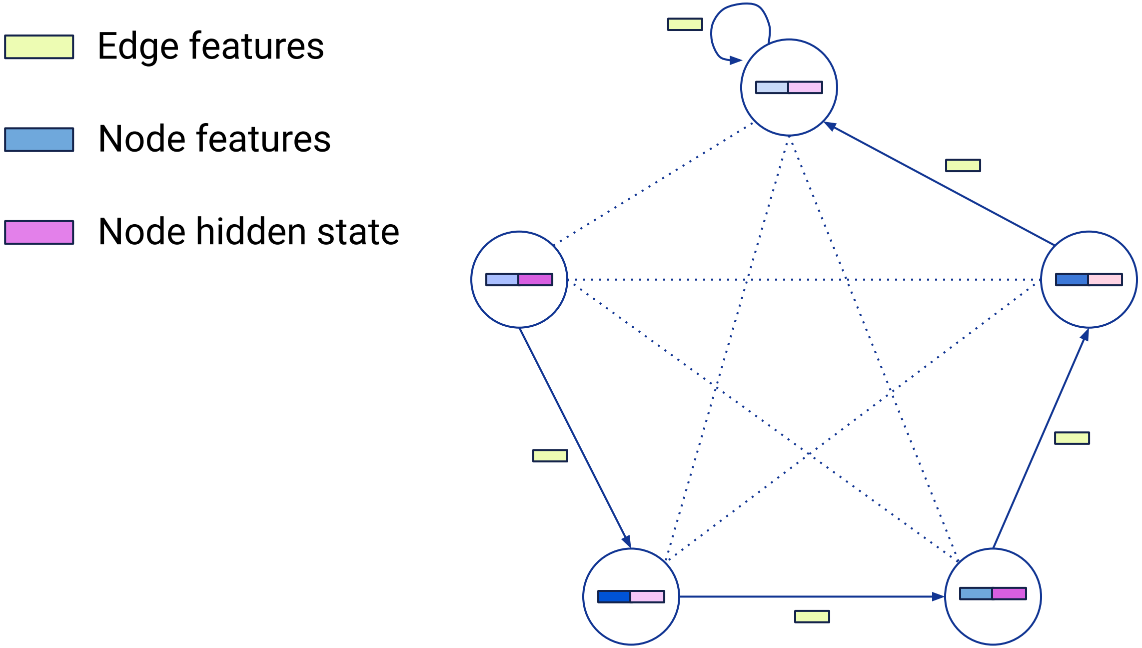
Representation: *processing*

Edge features

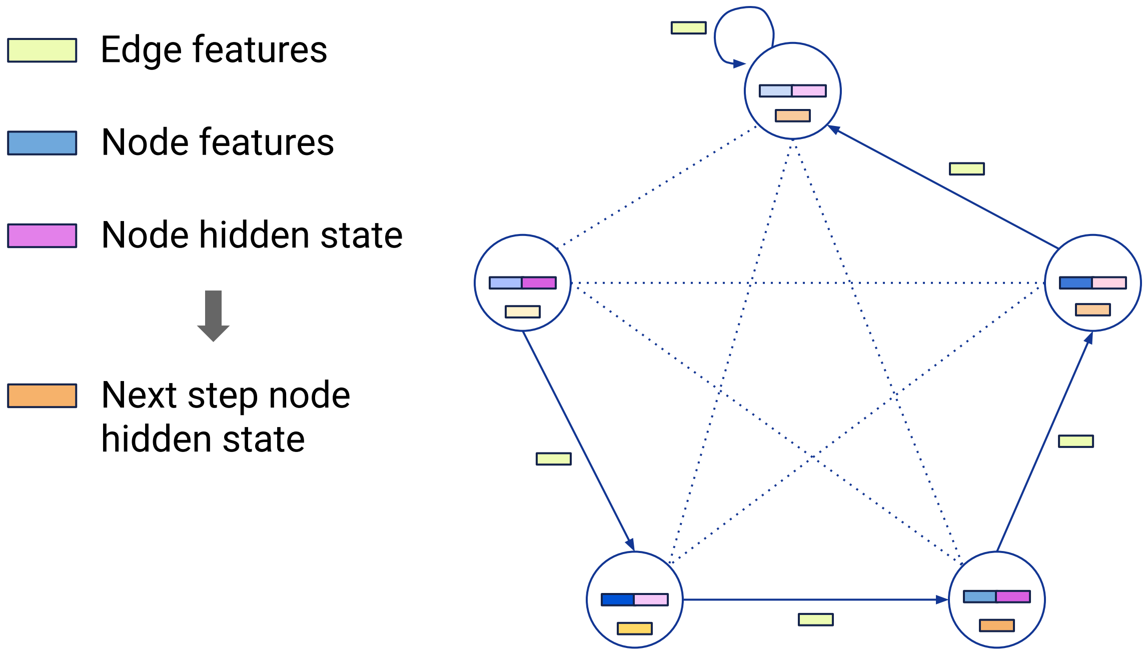
Node features



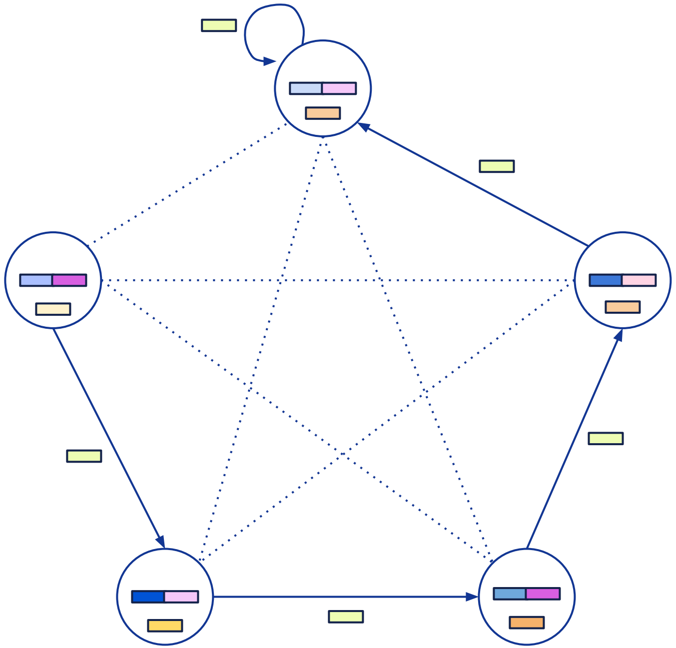
Representation: *processing*



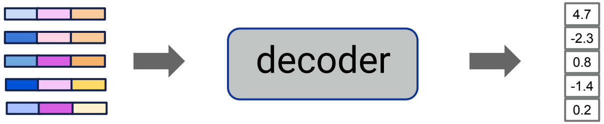
Representation: *processing*



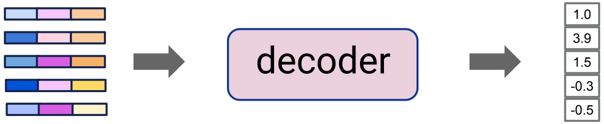
Representation: *decoding*



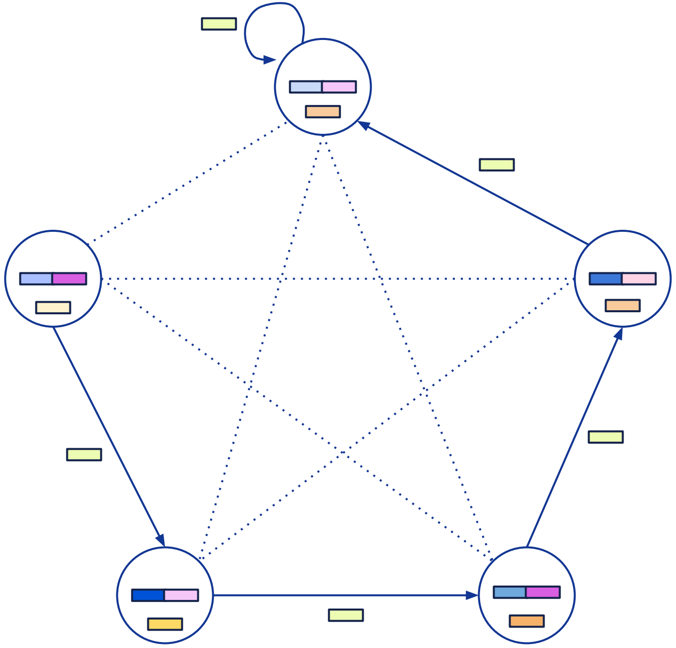
```
'i': (Stage.HINT, Location.NODE, Type.MASK_ONE)
```



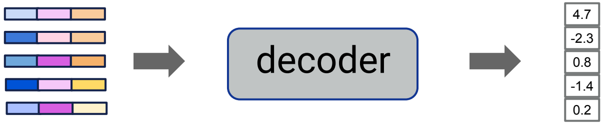

```
'j': (Stage.HINT, Location.NODE, Type.MASK_ONE)
```



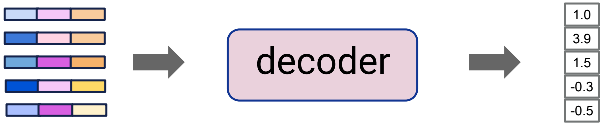
Representation: *decoding*



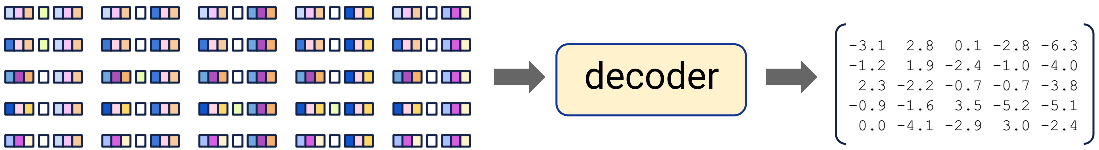
`'i': (Stage.HINT, Location.NODE, Type.MASK_ONE)`



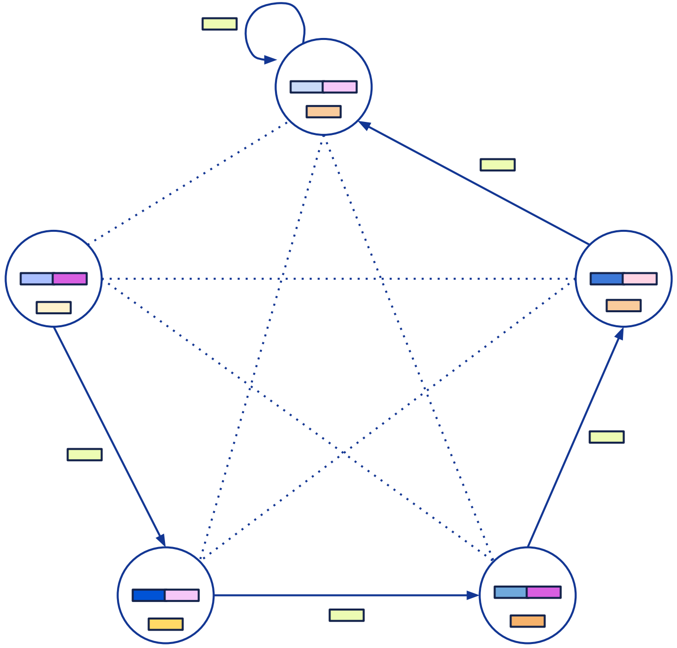
`'j': (Stage.HINT, Location.NODE, Type.MASK_ONE)`



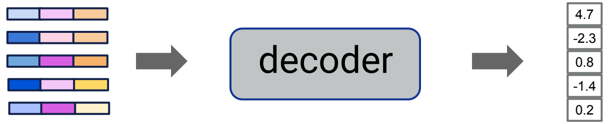
`'pred_h': (Stage.HINT, Location.NODE, Type.POINTER)`



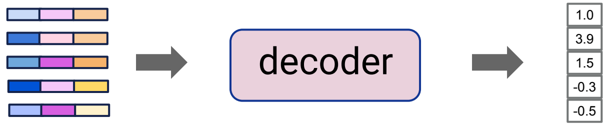
Representation: *decoding*



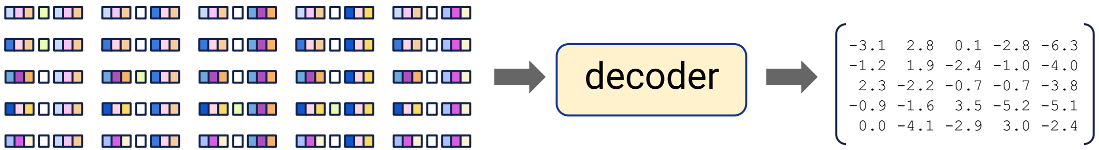
'i': (Stage.**HINT**, Location.**NODE**, Type.**MASK_ONE**)



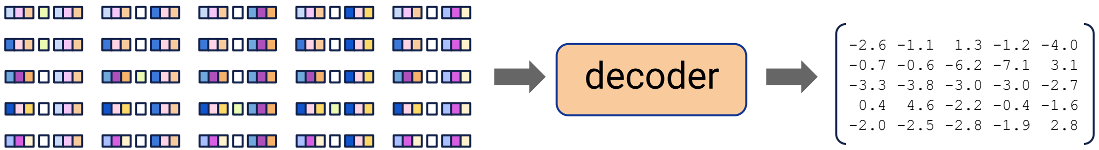
'j': (Stage.**HINT**, Location.**NODE**, Type.**MASK_ONE**)



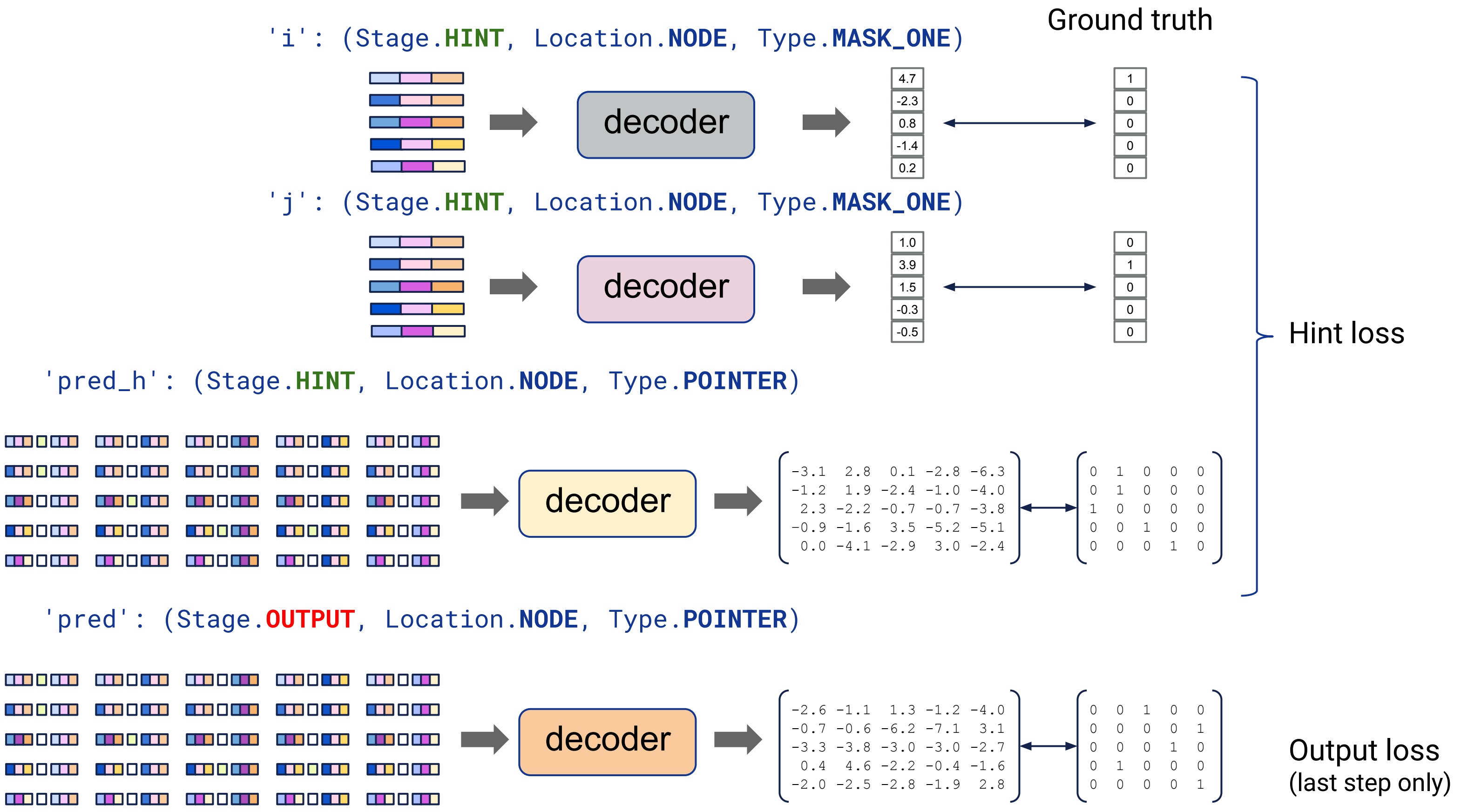
'pred_h': (Stage.**HINT**, Location.**NODE**, Type.**POINTER**)



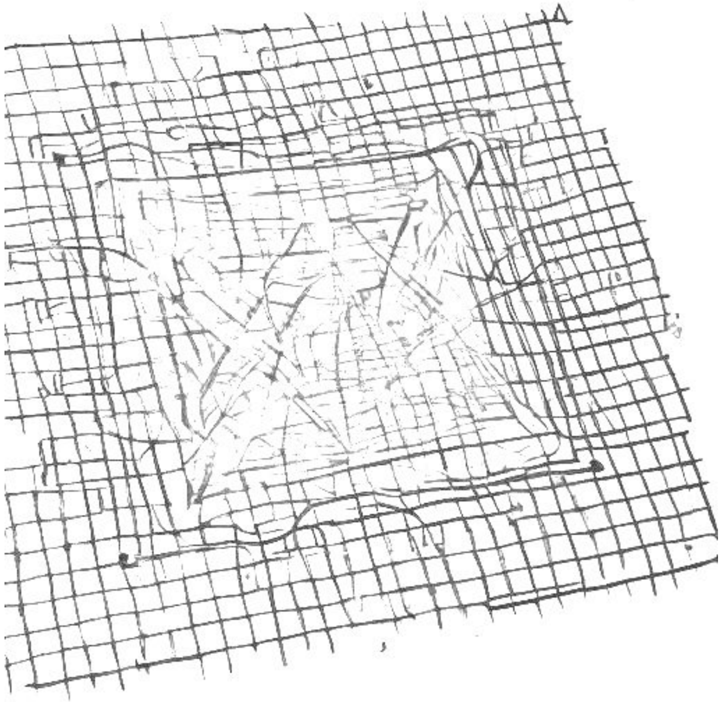
'pred': (Stage.**OUTPUT**, Location.**NODE**, Type.**POINTER**)



Training



Ground truth
'i': (Stage.HINT, Location.NODE, Type.MASK_ONE)
decoder
4.7
-2.3
0.8
-1.4
0.2
1
0
0
0
0
'j': (Stage.HINT, Location.NODE, Type.MASK_ONE)
decoder
1.0
3.9
1.5
-0.3
-0.5
0
1
0
0
0
Hint loss
'pred_h': (Stage.HINT, Location.NODE, Type.POINTER)
decoder
-3.1 2.8 0.1 -2.8 -6.3
-1.2 1.9 -2.4 -1.0 -4.0
2.3 -2.2 -0.7 -0.7 -3.8
-0.9 -1.6 3.5 -5.2 -5.1
0.0 -4.1 -2.9 3.0 -2.4
0 1 0 0 0
0 1 0 0 0
1 0 0 0 0
0 0 1 0 0
0 0 0 1 0
'pred': (Stage.OUTPUT, Location.NODE, Type.POINTER)
decoder
-2.6 -1.1 1.3 -1.2 -4.0
-0.7 -0.6 -6.2 -7.1 3.1
-3.3 -3.8 -3.0 -3.0 -2.7
0.4 4.6 -2.2 -0.4 -1.6
-2.0 -2.5 -2.8 -1.9 2.8
0 0 1 0 0
0 0 0 0 1
0 0 0 1 0
0 1 0 0 0
0 0 0 0 1
Output loss
(last step only)



Colab time!

Thank you!

Questions?

petarv@deepmind.com
https://petar-v.com