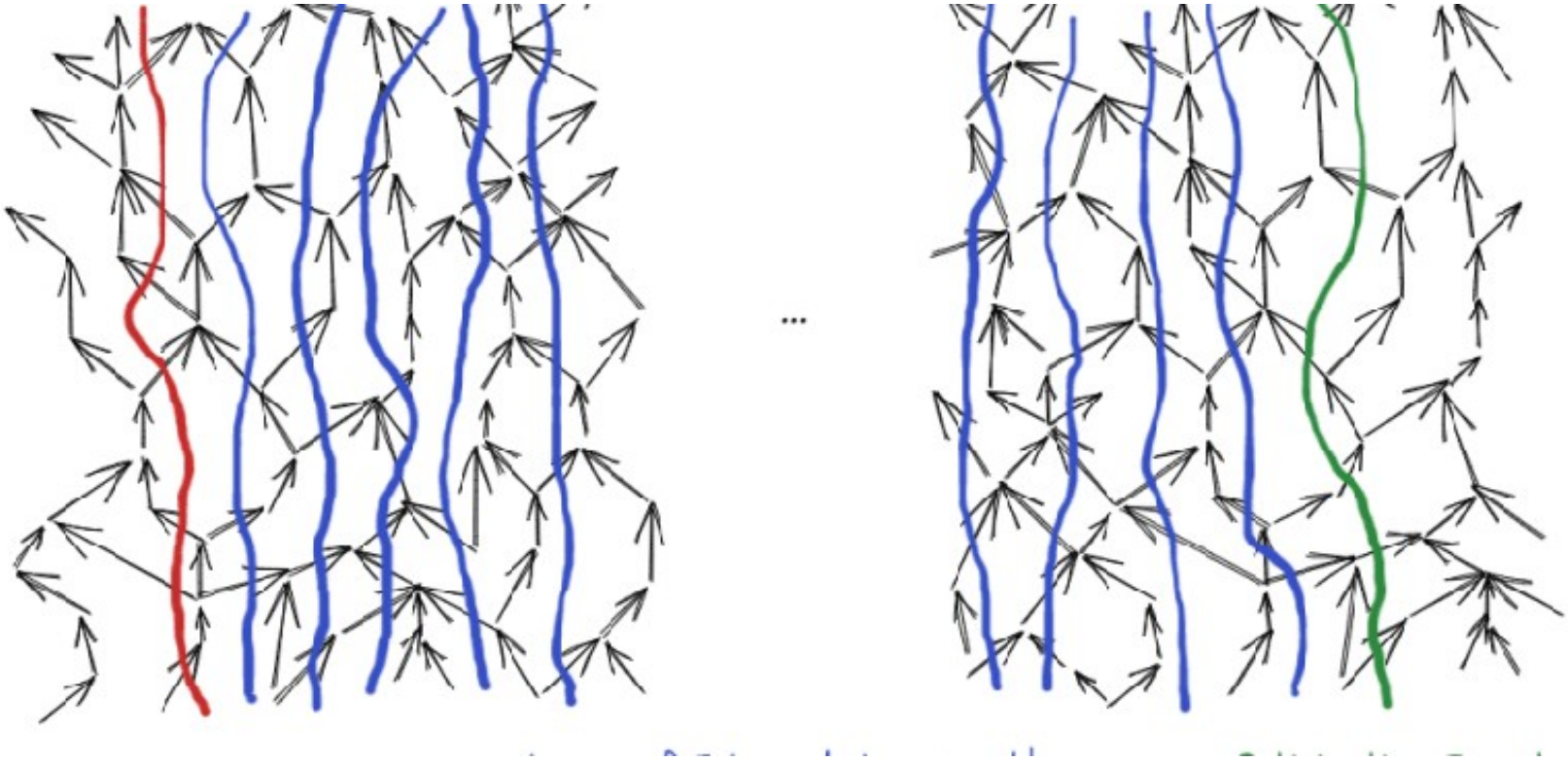


Basic Foundations for Agent Models

1. Against Time in Agent Models
2. Writing Causal Models Like We Write Programs
3. Utility Maximization = Description Length Minimization
4. Optimization at a Distance
5. Bits of Optimization Can Only Be Lost Over A Distance
6. The "Measuring Stick of Utility" Problem
7. Distributed Decisions
8. What's General-Purpose Search, And Why Might We Expect To See It In Trained ML Systems?

Against Time in Agent Models

Crossposted from the AI Alignment Forum. May contain more technical jargon than usual.

When programming distributed systems, we always have many computations running in parallel. Our servers handle multiple requests in parallel, perform read and write operations on the database in parallel, etc.

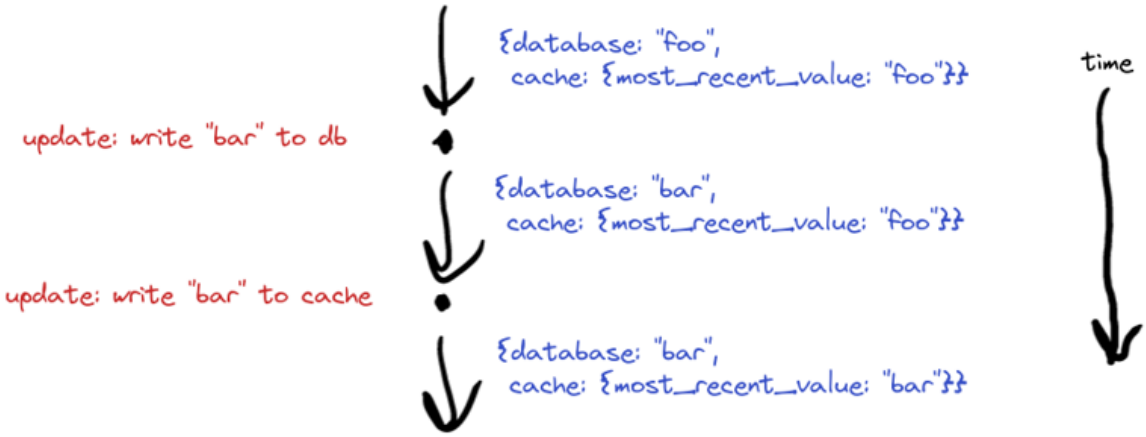
The prototypical headaches of distributed programming involve multiple processes running in parallel, each performing multiple read/write operations on the same database fields. Maybe some database field says “foo”, and process 1 overwrites it with “bar”. Process 2 reads the field - depending on the timing, it may see either “foo” or “bar”. Then process 2 does some computation and writes another field - for instance, maybe it sees “foo” and writes {“most_recent_value”: “foo”} to a cache. Meanwhile, process 1 overwrote “foo” with “bar”, so it also overwrites the cache with {“most_recent_value”: “bar”}. But these two processes are running in parallel, so these operations could happen in any order - including interleaving. For instance, the order could be:

1. Process 2 reads “foo”
2. Process 1 overwrites “foo” with “bar”
3. Process 1 overwrites the cache with {“most_recent_value”: “bar”}
4. Process 2 overwrites the cache with {“most_recent_value”: “foo”}

… and now the cached value no longer matches the value in the database; our cache is broken.

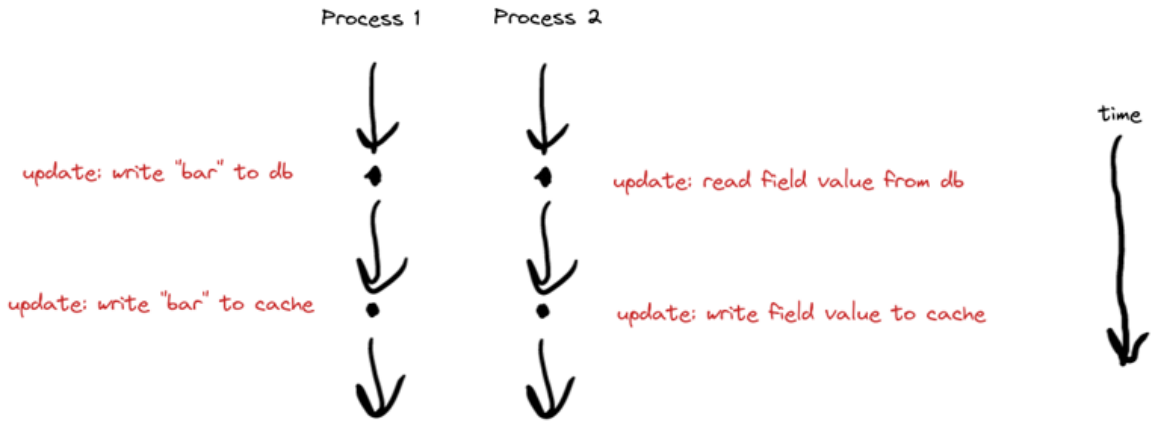
One of the main heuristics for thinking about this sort of problem in distributed programming is: **there is no synchronous time**. What does that mean?

Well, in programming we often picture a “state-update” model: the system has some state, and at each timestep the state is updated. The update rule is a well-defined function of the state; every update happens at a well-defined time. This is how each of the individual processes works in our example: each executes two steps in a well-defined order, and each step changes the state of the system

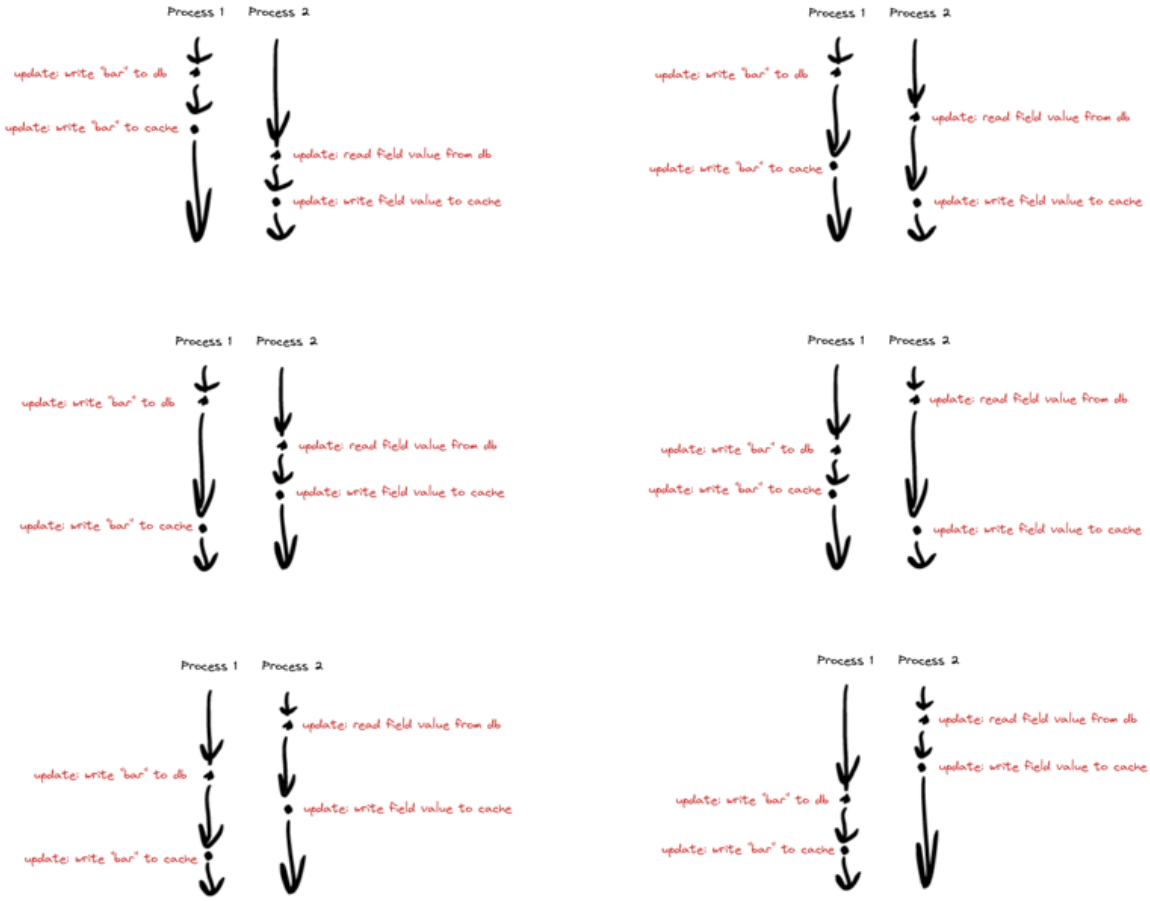


Single process: a well-defined sequence of steps, each updating the state.

But with multiple processes in parallel, this state-update model no longer works. In our example, we can diagram our two processes like this:



Each process has its own internal “time”: the database read/write happens first, and the cache overwrite happens second. But *between* processes, there is no guaranteed time-ordering. For instance, the first step of process 1 could happen before all of process 2, in between the steps of process 2, or after all of process 2.



Two processes: many possible time-orderings of the operations.

We cannot accurately represent this system as executing along one single time-dimension. Proof:

- Step 1 of process 1 is not guaranteed to happen either before or after step 1 of process 2; at best we could represent them as happening “at the same time”

- Step 2 of process 1 is also not guaranteed to happen either before or after step 1 of process 2; at best we could represent them as happening “at the same time”
- … but step 2 of process 1 *is* unambiguously *after* step 1 of process 1 in time, so the two steps can’t happen at the same time.

In order to accurately represent this sort of thing, it has to be possible for one step to be unambiguously after another, even though both of them are neither before nor after some third step.

The “most general” data structure to represent such a relationship is not a one-dimension “timeline” (i.e. total order), but rather a directed acyclic graph (i.e. partial order). That’s how time works in distributed systems: it’s a partial order, not a total order. A DAG, not a timeline. That DAG goes by many different names - including computation DAG, computation circuit, or causal model.

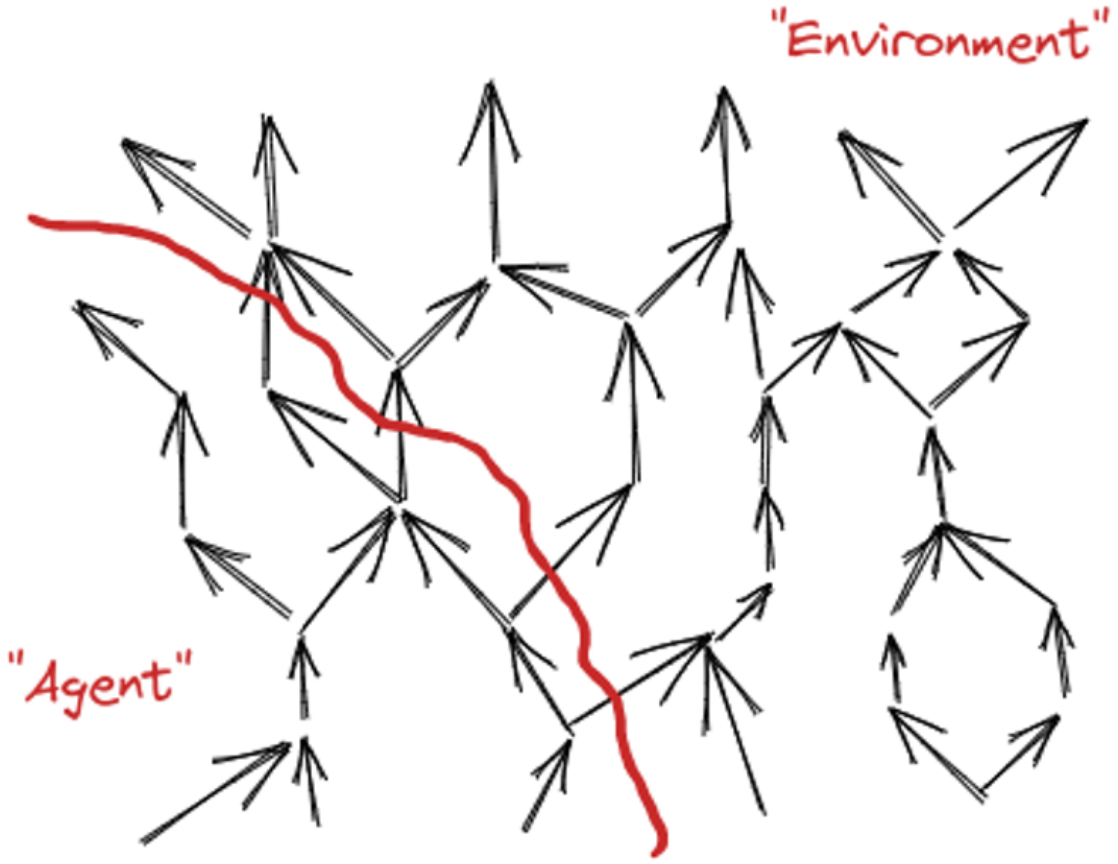
Beyond Distributed Programming

The same basic idea carries over to distributed systems more generally - i.e. any system physically spread out in space, with lots of different stuff going on in parallel. In a distributed system, “time” is a partial order, not a total order.

In the context of embedded agents: we want to model agenty systems which are “made of parts”, i.e. the agent is itself a system physically spread out in space with lots of different stuff going on in parallel. Likewise, the environment is made of parts. Both are distributed systems.

This is in contrast to state-update models of agency. In a state-update model, the environment has some state, the agent has some state, and at each timestep their states update. The update rule is a well-defined function of state; every update happens at a well-defined time.

Instead of the state-update picture, I usually picture an agent and its environment as a computation DAG (aka circuit aka causal model), where each node is a self-contained local computation. We carve off some chunk of this DAG to call “the agent”.



The obvious “Cartesian boundary” - i.e. the interface between agent and environment - is just a Markov blanket in the DAG (i.e. a cut which breaks the graph into two parts). That turns out to be not-quite-right, but it’s a good conceptual starting point.

Main takeaway: computational DAGs let us talk about agents without imposing a synchronous notion of "time" or "state updates", so we can play well with distributed systems.

Writing Causal Models Like We Write Programs

Crossposted from the AI Alignment Forum. May contain more technical jargon than usual.

Clunc

We'll start with a made-up programming language called Clunc. The distinguishing feature of clunc is that it combines classes and functions into a single type, called a clunc. It looks like this:

```
            quad = Clunc {
    x = 4
    constant = 3
    linear = 2*x + constant
    result = x*x + linear
}
```

We could then go to a terminal:

```
            >>> quad.result
27
>>> quad.linear
11
```

In order to use this clunc like a function, we apply the do() operator. For instance,

```
            >>> quad3 = do(quad, x=2)
```

… creates a new clunc which is just like quad, except that x is 2 rather than 4:

```
            >>> quad3
Clunc {
    x = 2
    constant = 3
    linear = 2*x + constant
    result = x*x + linear
}
```

When we query fields of quad3, they reflect the new x-value:

```
            >>> quad3.result
11
>>> quad3.linear
7
```

There's no designated "input" or "output"; we can use the do() operator to override any values we please. For instance

```
            >>> quad_zero_linear = do(quad, linear=0)
>>> quad_zero_linear
```

```
Clunc {
    x = 4
    constant = 3
    linear = 0
    result = x*x + linear
}
>>> quad_zero_linear.result
16
```

A few quick notes:

- Clunc is purely clunctional: everything is immutable, and each variable can only be written once within a clunc. No in-place updates.
- Clunc is lazy.
- Variables can be set randomly, e.g. “x = random.normal(0, 1)”.
- The do() operator creates a new clunc instance with the changes applied. If there are any random variables, they are re-sampled within the new clunc. If we want multiple independent samples of a randomized clunc M, then we can call do(M) (without any changes applied) multiple times.

To make this whole thing Turing complete, we need one more piece: recursion. Cluncs can “call” other cluncs, including themselves:

```
            factorial = Clunc {
    n = 4
    base_result = 1
    recurse_result = do(factorial, n=n-1).result
    result = (n == 0) ? base_result : n * recurse_result
}
```

… and that’s where things get interesting.

Causal Models

Hopefully the mapping from clunc to probabilistic causal models is obvious: any clunc with random variables in it is a typical Pearl-style causal DAG, and the do() operator works exactly like it does for causal models. The “clunc” is really a *model*, given by structural equations. The one big change is the possibility of recursion: causal models “calling” other models or other instances of themselves.

To get some practice with this idea, let’s build a reasonably-involved analogue model of a ripple-carry adder circuit.

We’ll start at the level of a NAND gate (levels below that involve equilibrium models, which would require a bunch of tangential explanation). We’ll assume that we have some model M_{NAND}, and we use $do(M_{NAND}, a = \ldots, b = \ldots).result$ to get the (noisy) output voltage of the NAND gate in terms of the input voltages a and b. Since we’re building an analogue model, we’ll be using actual voltages (including noise), not just their binarized values.

We'll take M_{NAND} as given (i.e. assume somebody else built that model). Building everything out of NAND gates directly is annoying, so we'll make an XOR as well:

M_{xor} = Model{

a = 0.0

b = 0.0

intermediate = do(M_{NAND}, a = a, b = b). result

left = do(M_{NAND}, a = a, b = intermediate). result

right = do(M_{NAND}, a = intermediate, b = b). result

result = do(M_{NAND}, a = left, b = right). result

}

This looks like a program which performs an XOR using NAND gates. But really, it's a Pearl-style causal DAG model which uses a lot of NAND-submodels. We can write out the joint probability distribution

$P[a = a^*, b = b^*, intermediate = i^*, left = l^*, right = r^*, result = result^* | M_{xor}]$ via the usual method, with each line in the model generating a term in the expansion:

$P[a = a^* | M_{XOR}] = I[a^* = 0]$

$P[b = b^* | M_{XOR}] = I[b^* = 0]$

$P[intermediate = i^* | M_{XOR}, a = a^*, b = b^*] = P[result = i^* | do(M_{NAND}, a = a^*, b = b^*)]$

$P[left = l^* | M_{XOR}, a = a^*, intermediate = i^*] = P[result = l^* | do(M_{NAND}, a = a^*, b = i^*)]$

$P[right = r^* | M_{XOR}, intermediate = i^*, b = b^*] = P[result = r^* | do(M_{NAND}, a = i^*, b = b^*)]$

$P[result = result^* | M_{XOR}, left = l^*, right = r^*] = P[result = result^* | do(M_{NAND}, a = l^*, b = r^*)]$

The full distribution is the product of those terms.

That's just the first step. Next, we need a full adder, a circuit block which computes the sum and carry bits for one "step" of binary long addition. It looks like this:

M_{full_adder} = Model{

$a = 0$

$b = 0$

$c = 0$

$s_{ab} = do(M_{XOR}, a = a, b = b).result$

$s = do(M_{XOR}, a = s_{ab}, b = c).result$

$carry_{ab} = do(M_{NAND}, a = a, b = b)$

$carry_c = do(M_{NAND}, a = s_{ab}, b = c)$

$carry = do(M_{NAND}, a = carry_{ab}, b = carry_c)$

}

As before, we can write out the components of the joint distribution line-by-line. I'll just do a few this time:

$P[a = a^*|M_{full\_adder}] = I[a^* = 0]$

. . .

$P[s_ab = \overset{*}{s}_{ab}|M_{full\_adder}, a = a^*, b = b^*] = P[result = \overset{*}{s}_{ab}|M_{XOR}, a = a^*, b = b^*]$

$P[s = s^*|M_{full\_adder}, s_{ab} = \overset{*}{s}_{ab}, c = c^*] = P[result = s^*|M_{XOR}, a = \overset{*}{s}_{ab}, b = c^*]$

. . .

Notice that some of these involve probabilities on the model M_{XOR}, which we could further expand using the joint distribution of M_{XOR} variables from earlier.

Finally, we can hook up a bunch of full adders to make our 32-bit ripple-carry adder:

$$M_{rc} = \text{Model}\{$$

$$a = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]$$

$$b = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]$$

$$f_0 = \text{do}(M_{\text{full\_adder}}, a = a[0], b = b[0])$$

$$f_i = \text{do}(M_{\text{full\_adder}}, a = a[i], b = b[i], c = f_{i-1}.\text{carry}) \quad \forall i \in 1, \ldots, 31$$

$$\text{result} = [f_0, \ldots, f_{31}]$$

$$\}$$

The components of the joint distribution for this one are left as an exercise for the reader.

Why Is This Useful?

Classes/functions let us re-use code; we don't have to repeat ourselves. Likewise, clunc-ish causal models let us re-use submodels; we don't have to repeat ourselves.

Obviously this has many of the same advantages as in programming. We can modularize our models, and fiddle with the internals of one submodel independently of other submodels. We can "subclass" our models via the do()-operator, to account for different contexts. Different people can work on different submodels independently - we could even imagine libraries of submodels. An electrical engineer could write a probabilistic causal model representing the low-level behavior of a chip; others could then import that model and use it as a reference when designing things which need to work with the chip, like packaging, accessories, etc.

From a more theoretical perspective, when we write programs with unbounded runtime, we *have* to have some way to re-use code: there's only so many lines in the program, so the program must visit some of the lines multiple times in the course of execution. Some lines must be re-used. Likewise for probabilistic models: if we want to define large models - including unbounded models - with small/finite definitions, then we need *some* way to re-use submodels. We could do that by writing things like "$\forall i :< \text{submodel}_i >$", but if we want Turing completeness anyway, we might as well go for recursion.

From a pure modelling perspective, the real world contains lots of repeating structures. If we're modelling things like cars or trees, we can re-use a lot of the information about one car when modelling another car. We think of cars as variations on a template, and that's exactly what the do() operator provides: we give it some "template" model, and apply modifications to it. The corresponding inverse problem then says: given a world full of things which are variations on some templates, find the templates and match them to the things - i.e. learn to recognize and model "cars" and "trees". Clunc-ish causal models are a natural fit for this sort of problem; they naturally represent things like "corvette with a flat tire".

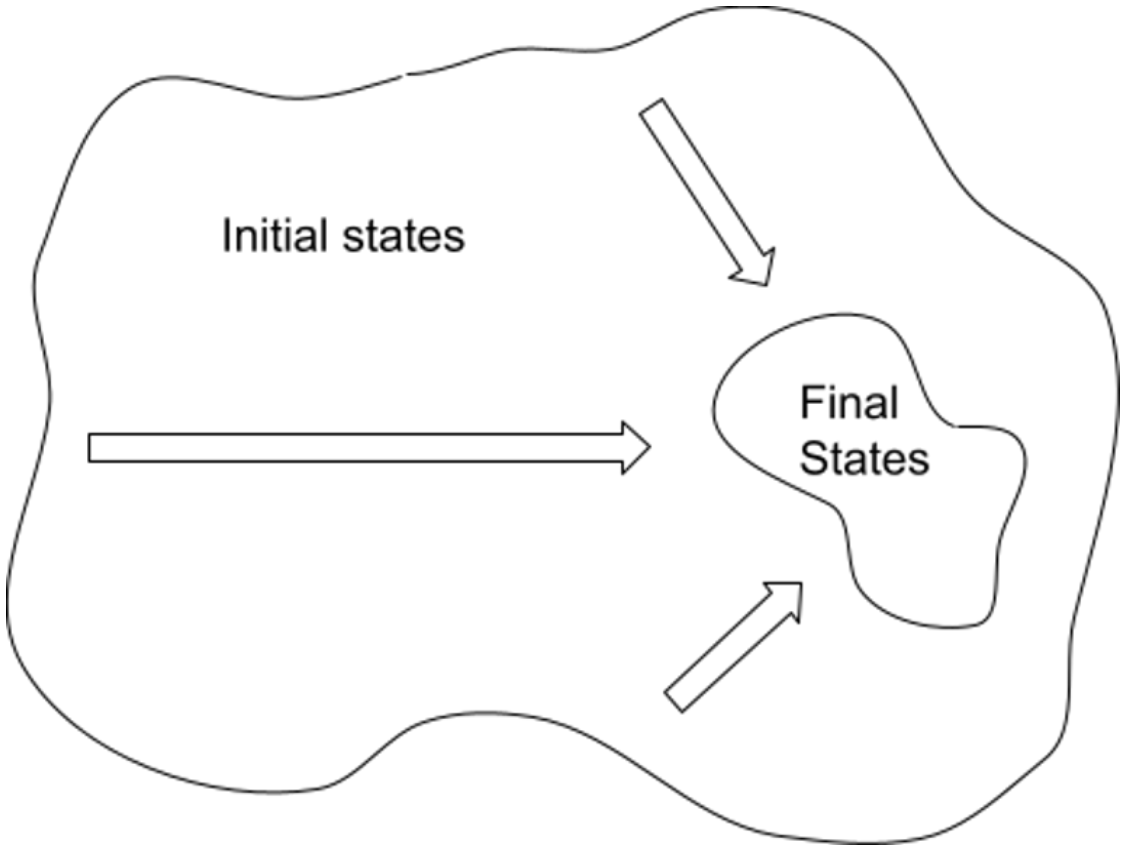
Finally, the main reason I've been thinking about this is to handle abstraction. Clunc-ish models make layers of abstraction natural; lower-level behaviors can be encapsulated in

submodels, just as we saw above with the ripple-carry adder. If we want to write abstraction-learning algorithms - algorithms which take in raw data and spit out multi-level models with layers of abstraction - then clunc-ish models are a natural form for their output. This is what multi-level world models look like.

Utility Maximization = Description Length Minimization

Crossposted from the AI Alignment Forum. May contain more technical jargon than usual.

There's a useful intuitive notion of "optimization" as pushing the world into a small set of states, starting from any of a large number of states. Visually:



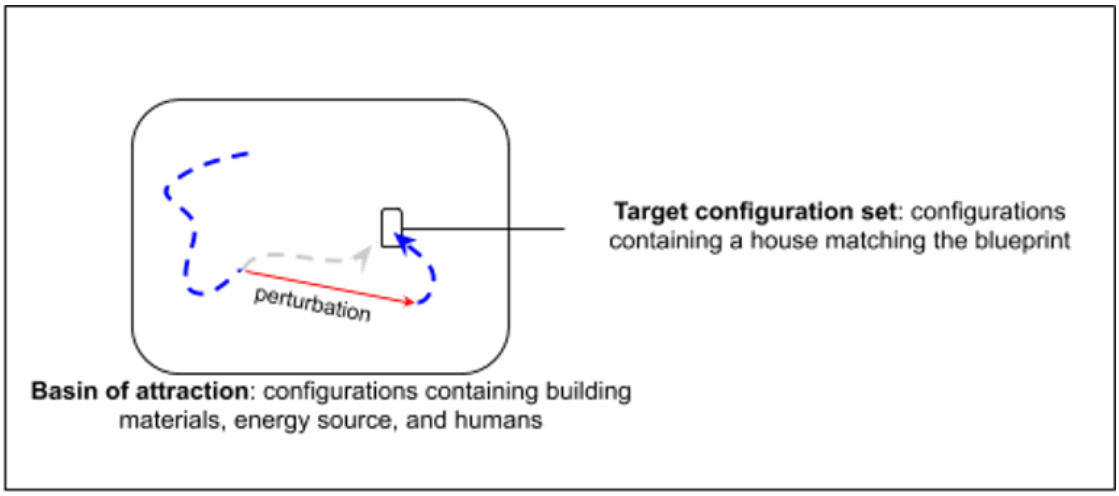
Yudkowsky and Flint both have notable formalizations of this "optimization as compression" idea.

This post presents a formalization of optimization-as-compression grounded in information theory. Specifically: **to "optimize" a system is to reduce the number of bits required to represent the system state using a particular encoding**. In other words, "optimizing" a system means making it compressible (in the information-theoretic sense) by a particular model.

This formalization turns out to be equivalent to expected utility maximization, and allows us to interpret any expected utility maximizer as "trying to make the world look like a particular model".

Conceptual Example: Building A House

Before diving into the formalism, we'll walk through a conceptual example, taken directly from Flint's Ground of Optimization: building a house. Here's Flint's diagram:



Configuration space: all possible configurations of matter within the chamber

The key idea here is that there's a wide variety of initial states (piles of lumber, etc) which all end up in the same target configuration set (finished house). The "perturbation" indicates that the initial state could change to some other state - e.g. someone could move all the lumber ten feet to the left - and we'd still end up with the house.

In terms of information-theoretic compression: we could imagine a model which says there is *probably* a house. Efficiently encoding samples from this model will mean using shorter bit-strings for world-states with a house, and longer bit-strings for world-states without a house. World-states with piles of lumber will therefore generally require more bits than world-states with a house. By turning the piles of lumber into a house, we reduce the number of bits required to represent the world-state using this particular encoding/model.

If that seems kind of trivial and obvious, then you've probably understood the idea; later sections will talk about how it ties into other things. If not, then the next section is probably for you.

Background Concepts From Information Theory

The basic motivating idea of information theory is that we can represent information using fewer bits, on average, if we use shorter representations for states which occur more often. For instance, Morse code uses only a single bit (".") to represent the letter "e", but four bits ("- - . -") to represent "q". This creates a strong connection between probabilistic models/distributions and optimal codes: a code which requires minimal average bits for one distribution (e.g. with lots of e's and few q's) will not be optimal for another distribution (e.g. with few e's and lots of q's).

For any random variable X generated by a probabilistic model M, we can compute the minimum average number of bits required to represent X. This is Shannon's famous entropy formula

$$-\sum_X P[X \mid M] \log P[X \mid M]$$

Assuming we're using an optimal encoding for model M, the number of bits used to encode a *particular* value x is $\log P[X = x|M]$. (Note that this is sometimes not an integer! Today we have algorithms which encode many samples at once, potentially even from different models/distributions, to achieve asymptotically minimal bit-usage. The "rounding error" only happens once for the whole collection of samples, so as the number of samples grows, the rounding error per sample goes to zero.)

Of course, we could be *wrong* about the distribution - we could use a code optimized for a model M_2 which is different from the "true" model M_1. In this case, the average number of bits used will be

$$-\sum_X P[X \mid M_1] \log P[X \mid M_2] = E[\log P[X \mid M_2] \mid M_1]$$

In this post, we'll use a "wrong" model M_2 intentionally - not because we *believe* it will yield short encodings, but because we *want* to push the world into states with short M_2-encodings. The model M_2 serves a role analogous to a utility function. Indeed, we'll see later on that every model M_2 is equivalent to a utility function, and vice-versa.

Formal Statement

Here are the variables involved in "optimization":

- World-state random variables X
- Parameters θ, θ' which will be optimized
- Probabilistic world-model $M_1(\theta)$ representing the distribution of X
- Probabilistic world-model M_2 representing the encoding in which we wish to make X more compressible

An "optimizer" takes in some parameter-values θ, and returns new parameter-values θ' such that

$$E[-\log P[X \mid M_2] \mid M_1(\theta')] \leq E[-\log P[X \mid M_2] \mid M_1(\theta)]$$

... with equality if-and-only-if θ already achieves the smallest possible value. In English: we choose θ' to reduce the average number of bits required to encode a sample from $M_1(\theta')$, using a code optimal for M_2. This is essentially just our formula from the previous section for the number of bits used to encode a sample from M_1 using a code optimal for M_2.

Other than the information-theory parts, the main thing to emphasize is that we're mapping one parameter-value θ to a "more optimal" parameter-value θ'. This should work for many different "initial" θ-values, implying a kind of robustness to changes in θ. (This is roughly the same concept which Flint captured by talking about "perturbations" to the system-state.) In the context of iterative optimizers, our definition corresponds to one step of optimization; we could of course feed θ' back into the optimizer and repeat. We could even do this without having any distinguished "optimizer" subsystem - e.g. we might just have some dynamical system in which θ is a function of time, and successive values of θ_t satisfy the inequality condition.

Finally, note that our model M_1 is a function of θ. This form is general enough to encompass all the usual decision theories. For instance, under EDT, $M_1(\theta)$ would be some base model M conditioned on the data θ. Under CDT, $M_1(\theta)$ would instead be a causal intervention on a base model M, i.e. $M_1(\theta) = do(M, \Theta = \theta)$.

Equivalence to Expected Utility Optimization

Obviously our expression $E[-\log P[X|M_2]|M_1(\theta)]$ can be expressed as an expected utility: just set $u(X) = \log P[X|M_2]$. The slightly more interesting claim is that we can always go the other way: for any utility function $u(X)$, there is a corresponding model M_2, such that maximizing expected utility $u(X)$ is equivalent to minimizing expected bits to encode X using M_2.

The main trick here is that we can always add a constant to $u(X)$, or multiply $u(X)$ by a positive constant, and it will still "be the same utility" - i.e. an agent with the new utility will always make the same choices as the old. So, we set

$$\alpha\, u(X) + \beta = \log P[X|M_2] \implies P[X|M_2] = e^{\beta} e^{\alpha u(X)}$$

... and look for α, β which give us a valid probability distribution (i.e. all probabilities are nonnegative and sum to 1).

Since everything is in an exponent, all our probabilities will be nonnegative for any α, β, so that constraint is trivially satisfied. To make the distribution sum to one, we simply set $\beta = -\ln \sum_X e^{\alpha u(X)}$. So, not only can we find a model M_2 for any $u(X)$, we actually find a whole family of them - one for each $\alpha > 0$.

(This also reveals a degree of freedom in our original definition: we can always create a new model M_2 with $P[X|M_2] = \frac{1}{Z} P[X|M_2]^{\alpha}$ without changing the behavior.)

So What Does This Buy Us?

If this formulation is equivalent to expected utility maximization, why view it this way?

Intuitively, this view gives more semantics to our "utility functions". They have built-in "meanings"; they're not just preference orderings.

Mathematically, the immediately obvious step for anyone with an information theory background is to write:

$$E[-\log P[X|M_2]|M_1] = -\sum_X P[X|M_1] \log P[X|M_1] + P[X|M_1] \log \frac{P[X|M_2]}{P[X|M_1]}$$

$$= H(X|M_1) + D_{KL}(M_2.X||M_1.X)$$

The expected number of bits required to encode X using M_2 is the entropy of X plus the Kullback-Liebler divergence of (distribution of X under model M_2) from (distribution of X under model M_1). Both of those terms are nonnegative. The first measures "how noisy" X is, the second measures "how close" the distributions are under our two models.

Intuitively, this math says that we can decompose the objective $E[-\log P[X|M_2]|M_1]$ into two pieces:

- Make X more predictable
- Make the distribution of X "close to" the distribution $P[X|M_2]$, with closeness measured by KL-divergence

Combined with the previous section: we can take any expected utility maximization problem, and decompose it into an entropy minimization term plus a "make-the-world-look-like-this-specific-model" term.

This becomes especially interesting in situations where the entropy of X cannot be reduced - e.g. thermodynamics. If the entropy H(X) is fixed, then only the KL-divergence term remains. In this case, we can directly interpret the optimization problem as "make the world-state distribution look like $P[X|M_2]$". If we started from an expected utility optimization problem, then we derive a model M_2 such that optimizing expected utility is equivalent to making the world look as much as possible like M_2.

In fact, even when H(X) is not fixed, we can build equivalent models M_1', M_2' for which it is fixed, by adding new variables to X. Suppose, for example, that we can choose between flipping a coin and rolling a die to determine X_0. We can change the model so that both the coin flip and the die roll always happen, and we include their outcomes in X. We then choose whether to set X_0 equal to the coin flip result or the die roll result, but in either case the entropy of X is the same, since both are included. M_2' simply ignores all the new components added to X (i.e. it implicitly has a uniform distribution on the new components).

So, starting from an expected utility maximization problem, we can transform to an equivalent minimum coded bits problem, and from there to an equivalent minimum KL-divergence problem. We can then interpret the optimization as "choose θ to make $M_1(\theta)$ as close as possible to M_2", with closeness measured by KL-divergence.

What I Imagine This Might Be Useful For

In general, interpretations of probability grounded in information theory are much more solid than interpretations grounded in coherence theorems. However, information-theoretic groundings *only* talk about probability, not about "goals" or "agents" or anything utility-like. Here, we've transformed expected utility maximization into something explicitly information-theoretic and conceptually natural. This seems like a potentially-promising step toward better foundations of agency. I imagine there's probably purely-information-theoretic "coherence theorems" to be found.

Another natural direction to take this in is thermodynamic connections, e.g. combining it with a generalized heat engine. I wouldn't be surprised if this also tied in with information-theoretic "coherence theorems" - in particular, I imagine that negentropy could serve as a universal "resource", replacing the "dollars" typically used as a measuring stick in coherence theorems.

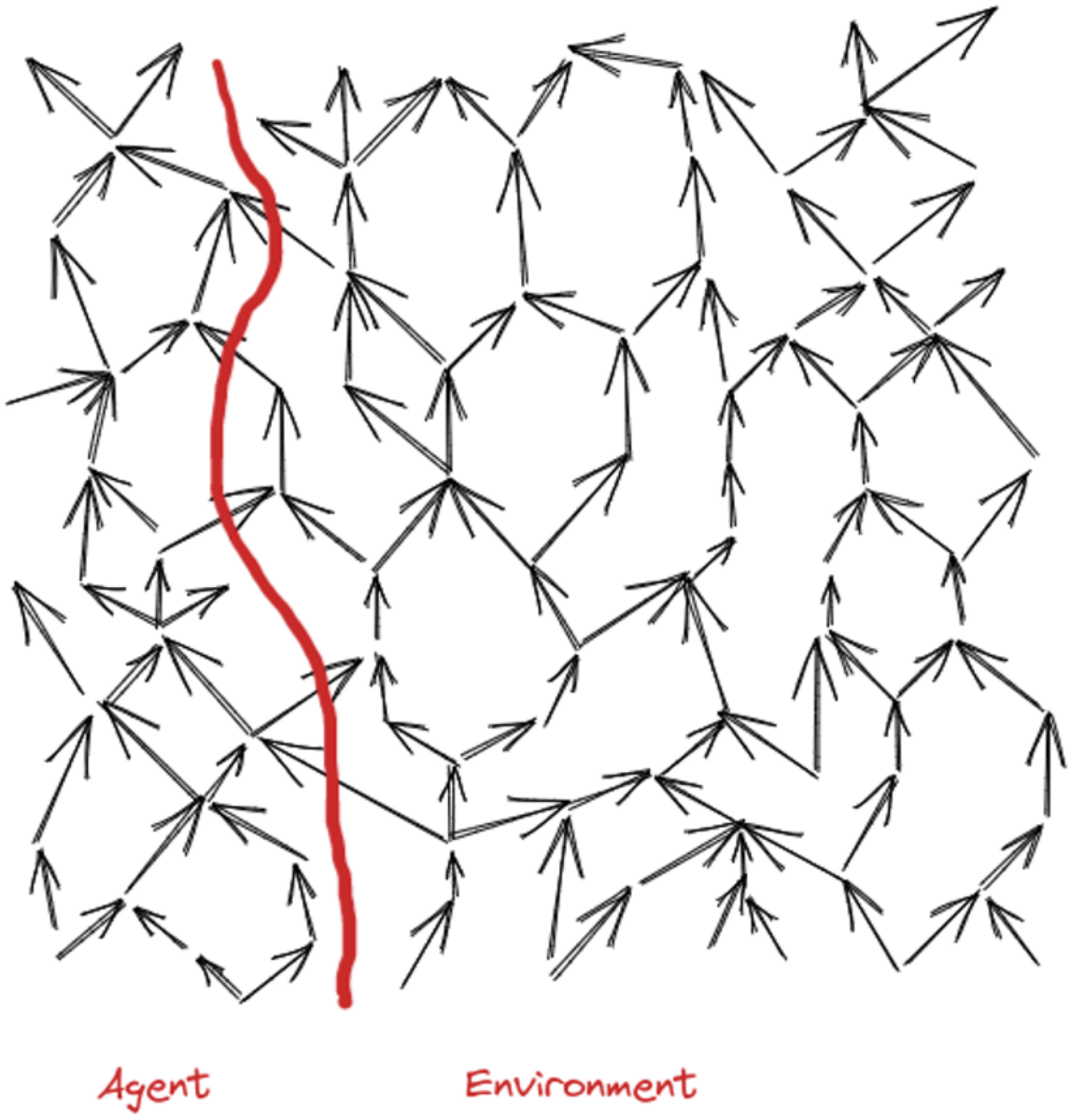
Overall, the whole formulation smells like it could provide foundations much more amenable to embedded agency.

Finally, there's probably some nice connection to predictive processing. In all likelihood, Karl Friston has already said all this, but it has yet to be distilled and disseminated to the rest of us.

Optimization at a Distance

Crossposted from the AI Alignment Forum. May contain more technical jargon than usual.

We have a computational graph (aka circuit aka causal model) representing an agent and its environment. We've chosen a cut through the graph to separate "agent" from "environment" - i.e. a Cartesian boundary. Arrows from environment to agent through the boundary are "observations"; arrows from agent to environment are "actions".



Presumably the agent is arranged so that the "actions" optimize something. The actions "steer" some nodes in the system toward particular values.

Let's highlight a few problems with this as a generic agent model...

Microscopic Interactions

My human body interfaces with the world via the entire surface area of my skin, including molecules in my hair randomly bumping into air molecules. All of those tiny interactions are arrows going through the supposed “Cartesian boundary” around my body. These don’t intuitively seem like “actions” or “observations”, at least not beyond some high-level observations of temperature and pressure.

In general, low-level boundaries will have lots of tiny interactions crossing them which don’t conceptually seem like “actions” or “observations”.

Flexible Boundaries

When I’m driving, I often identify with the car rather than with my body. Or if I lose a limb, I stop identifying with the lost limb. (Same goes for using the toilet - I’ve heard that it’s quite emotionally stressful for children during potty training to throw away something which came from their physical body, because they still identify with it.)

In general, it’s ambiguous what Cartesian boundary to use; our conceptual boundaries around an “agent” don’t seem to correspond perfectly to any particular physical surface.

An Agent Optimizing Its Own Actions

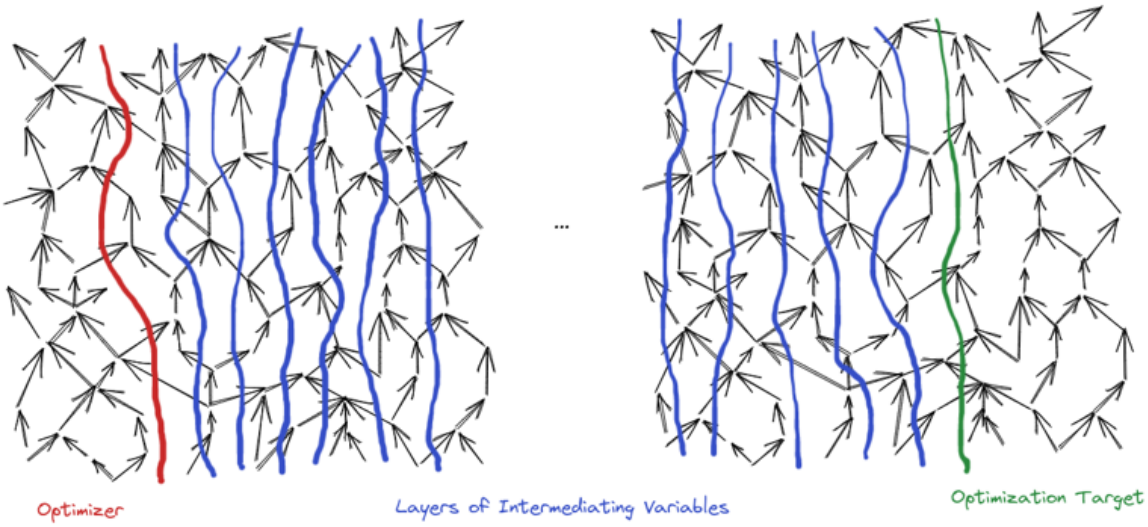
I could draw a supposed “Cartesian boundary” around a rock, and declare all the interactions between the rock and its environment to be “actions” and “observations”. If someone asks what the rock is optimizing, I’ll say “the actions” - i.e. the rock “wants” to do whatever it is that the rock in fact does.

In general, we intuitively conceive of “agents” as optimizers in some nontrivial sense. Optimizing actions doesn’t cut it; we generally don’t think of something as an agent unless it’s optimizing something out in the environment away from itself.

Solution: Optimization At A Distance

Let’s solve all of these problems in one fell swoop.

We’ll start with the rock problem. One natural answer is to declare that we’re only interested in agents which optimize things “far away” from themselves. What does that mean? Well, as long as we’re already representing the world as a computational DAG, we might as well say that two chunks of our computation DAG are “far apart” when there are many intermediating layers between them. Like this:



If you've read the Telephone Theorem post, it's the same idea.

For instance, if I'm planning a party, then the actions I take now are far away in time (and probably also space) from the party they're optimizing. The "intermediate layers" might be snapshots of the universe-state at each time between the actions and the party. (... or they might be something else; there are usually many different ways to draw intermediate layers between far-apart things.)

This applies surprisingly well even in situations like reinforcement learning, where we don't typically think of the objective as "far away" from the agent. If I'm a reinforcement learner optimizing for some reward I'll receive later, that later reward is still typically far away *from my current actions* . My actions impact the reward via some complicated causal path through the environment, acting through many intermediate layers.

So we've ruled out agents just "optimizing" their own actions. How does this solve the other two problems?

Abstract Summaries

We're using the same kind of model and the same notion of "far apart" as the Telephone Theorem, so we can carry that theorem over. The main takeaway is that far apart things interact only via a typically-relatively-small "abstract summary". This summary consists of the information which is arbitrarily well conserved as it propagates through the intermediate layers.

Because the agent only interacts with the far away things-it's-optimizing via a relatively-small summary, it's natural to define the "actions" and "observations" as the *contents of the summary* flowing in either direction, rather than all the low-level interactions flowing through the agent's supposed "Cartesian boundary". That solves the microscopic interactions problem: all the random bumping between my hair/skin and air molecules mostly doesn't impact things far away, except via a few summary variables like temperature and pressure.

This redefinition of "actions" and "observations" also makes the Cartesian boundary flexible. The Telephone Theorem says that the abstract summary consists of information which is arbitrarily well conserved as it propagates through the intermediate layers. So, the summary isn't very sensitive to *which* layer we declare to be "the Cartesian boundary"; we can move the boundary around quite a bit without changing the abstract "agent" we're talking about. (Though obviously if we move the Cartesian boundary to some totally different part of the

world, that may change what “agent” we’re talking about.) If we want, we could even stop thinking of the boundary as localized to a particular cut through the graph at all.

Aside: Dynamic Programming

When Adam Shimi first suggested to me a couple years ago that “optimization far away” might be important somehow, one counterargument I raised was dynamic programming (DP): if the agent is optimizing an expected utility function over something far away, then we can use DP to propagate the expected utility function back through the intermediate layers to find an equivalent utility function over the agent’s actions:

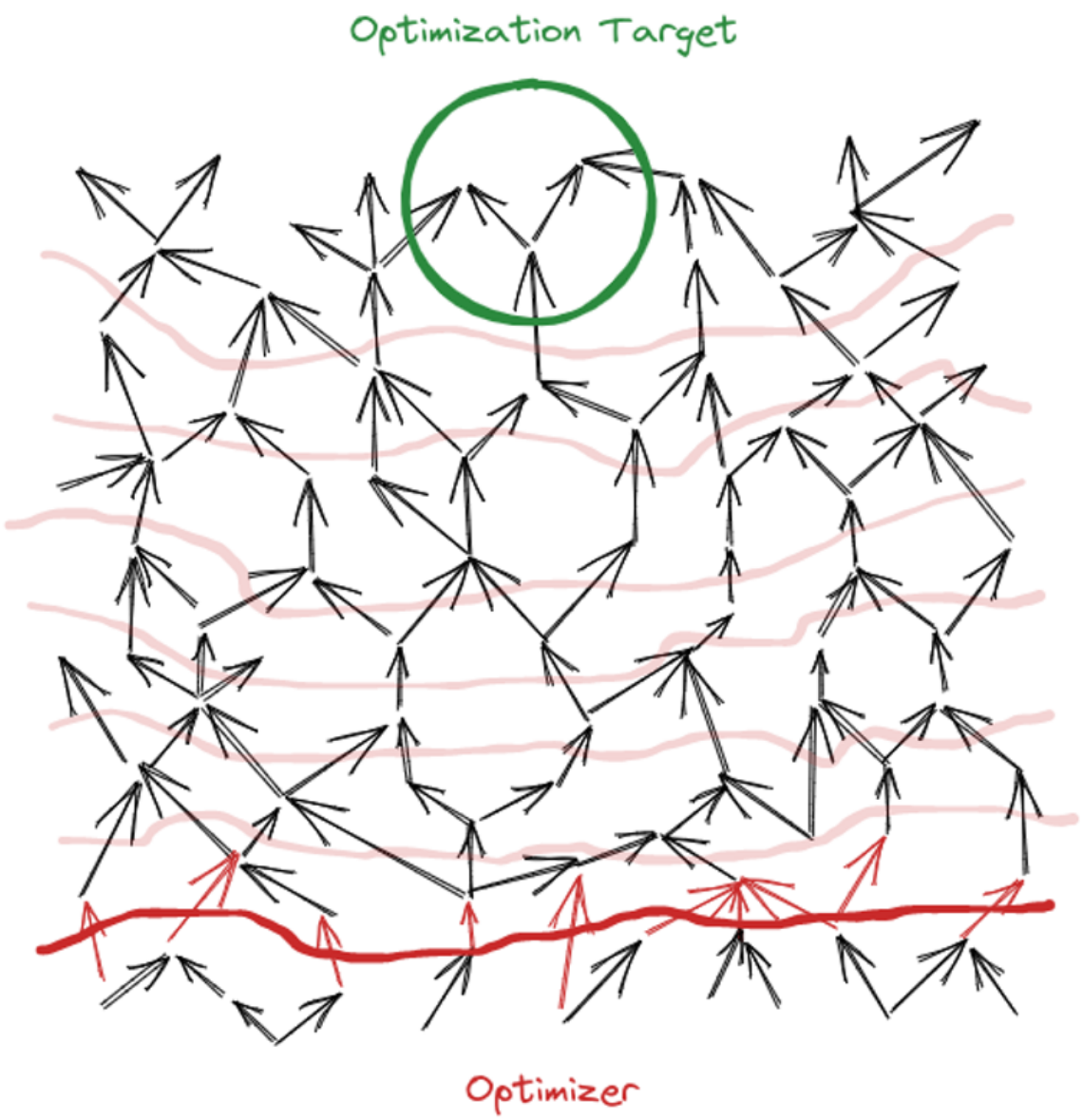
$$u'(A) = E[u(X) \mid do(A)]$$

This isn’t actually a problem, though. It says that optimization far away is equivalent to some optimization nearby. But the reverse does not necessarily hold: optimization nearby is not necessarily equivalent to some optimization far away. This makes sense: optimization nearby is a trivial condition which matches basically any system, and therefore will match the interesting cases as well as the uninteresting cases.

(Note that I haven’t actually demonstrated here that optimization at a distance *is* nontrivial, i.e. that some systems do optimize at a distance and others don’t; I’ve just dismissed one possible counterargument. I have several posts planned on optimization at a distance over the next few weeks, and nontriviality will be in one of them.)

Mental Picture

I like to picture optimization at a distance like a satellite dish or phased array:



Lots of little “actions” produce a strong coherent influence, which can propagate far away to impact the optimization target.

In a phased array, lots of little antennas distributed over an area are all controlled simultaneously, so that their waves add up to one big coherent wave which can propagate over a long distance. Optimization at a distance works the same way: there’s lots of little actions distributed over space/time, all controlled in such a way that their influence can add up coherently and propagate over a long distance to optimize some far-away target.

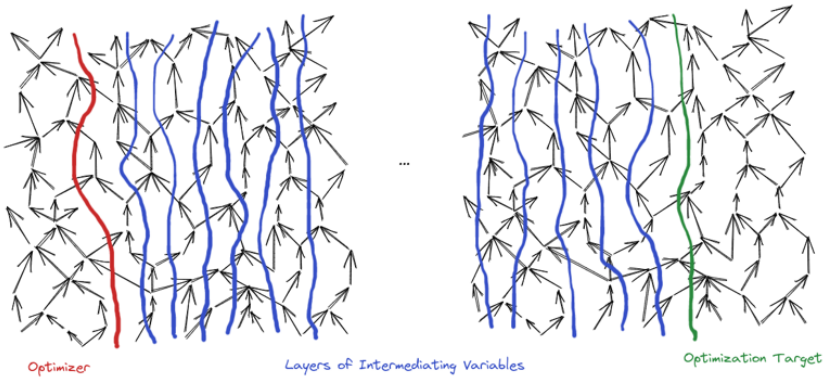
Bits of Optimization Can Only Be Lost Over A Distance

Crossposted from the AI Alignment Forum. May contain more technical jargon than usual.

When we think of "optimization" as compressing some part of the universe into a relatively small number of possible states, it's very natural to quantify that compression in terms of "bits of optimization". Example: we have a marble which could be in any of 16 different slots in a box (assume/approximate uniform probability on each). We turn the box on its side, shake it, and set it down so the marble will end up in one of the 4 slots on the downward side (again, assume/approximate uniform probability on each). Then we've compressed the marble-state from 16 possibilities to 4, cut the state-space in half twice, and therefore performed two bits of optimization.

In the language of information theory, this quantity is the KL-divergence between the initial and final distributions.

In this post, we'll prove our first simple theorem about optimization at a distance: the number of bits of optimization applied can only decrease over a distance. In particular, in our optimization at a distance picture:



… the number of bits of optimization applied to the far-away optimization target cannot be any larger than the number of bits of optimization applied to the optimizer's direct outputs.

The setup: first, we'll need two distributions to compare over the optimizer's direct outputs. You might compare the actual output-distribution to uniform randomness, or independent outputs. If the optimizer is e.g. a trained neural net, you might compare its output-distribution to the output-distribution of a randomly initialized net. If the optimizer has some sort of explicit optimization loop (like e.g. gradient descent), then you might compare its outputs to the initial outputs tested in that loop. These all have different interpretations and applications; the math here will apply to all of them.

Let's name some variables in this setup:

- Optimizer's direct outputs: A (for "actions")
- i'th intermediating layer: M_i (with $M_0 = A$)
- Reference distribution over everything: $P[A, M|ref]$
- Actual distribution over everything: $P[A, M|opt]$

By assumption, only the optimizer differs between the reference and actual distributions; the rest of the Bayes net is the same. Mathematically, that means $P[M_{i+1}|M_i, opt] = P[M_{i+1}|M_i, ref] =: P[M_{i+1}|M_i]$ (and of course both distributions factor over the same underlying graph).

Once we have two distributions over the optimizer's direct outputs, they induce two distributions over each subsequent layer of intermediating variables, simply by propagating through each layer:

$$P[M_{i+1}|ref] = \sum_{M_i} P[M_{i+1}|M_i] P[M_i|ref]$$

$$P[M_{i+1}|opt] = \sum_{M_i} P[M_{i+1}|M_i] P[M_i|opt]$$

At each layer, we can compute the number of bits of optimization applied to that layer, i.e. how much that layer's state-space is compressed by the actual distribution relative to the reference distribution. That's the KL-divergence between the distributions: $D_{KL}(P[M_i|opt]||P[M_i|ref])$.

To prove our theorem, we just need to show that $D_{KL}(P[M_{i+1}|opt]||P[M_{i+1}|ref]) \leq D_{KL}(P[M_i|opt]||P[M_i|ref])$. To do that, we'll use the chain rule of KL divergence to expand $D_{KL}(P[M_i, M_{i+1}|opt]||P[M_i, M_{i+1}|ref])$ in two different ways. First:

$$D_{KL}(P[M_i, M_{i+1}|opt]||P[M_i, M_{i+1}|ref]) = D_{KL}(P[M_i|opt]||P[M_i|ref]) + D_{KL}(P[M_{i+1}|M_i, opt]$$

Recall that $P[M_{i+1}|M_i, opt]$ and $P[M_{i+1}|M_i, ref]$ are the same, so $D_{KL}(P[M_{i+1}|M_i, opt]||P[M_{i+1}|M_i, ref]) = 0$, and our first expression simplifies to $D_{KL}(P[M_i, M_{i+1}|opt]||P[M_i, M_{i+1}|ref]) = D_{KL}(P[M_i|opt]||P[M_i|ref])$. Second:

$$D_{KL}(P[M_i, M_{i+1}|opt]||P[M_i, M_{i+1}|ref]) = D_{KL}(P[M_{i+1}|opt]||P[M_{i+1}|ref]) + D_{KL}(P[M_i|M_{i+1},$$

KL-divergence is always nonnegative, so we can drop the second term above and get an inequality:
$D_{KL}(P[M_i, M_{i+1}|opt]||P[M_i, M_{i+1}|ref]) \geq D_{KL}(P[M_{i+1}|opt]||P[M_{i+1}|ref])$

Now we just combine these two expressions for $D_{KL}(P[M_i, M_{i+1}|opt]||P[M_i, M_{i+1}|ref])$ and find

$$D_{KL}(P[M_i|opt]||P[M_i|ref]) \geq D_{KL}(P[M_{i+1}|opt]||P[M_{i+1}|ref])$$

… which is what we wanted to prove.

So: if we measure the number of bits of optimization applied to the optimizer's direct output, or to any particular layer, that provides an upper bound on the number of bits of optimization applied further away.

The "Measuring Stick of Utility" Problem

Crossposted from the AI Alignment Forum. May contain more technical jargon than usual.

Let's start with the simplest coherence theorem: suppose I'll pay to upgrade pepperoni pizza to mushroom, pay to upgrade mushroom to anchovy, and pay to upgrade anchovy to pepperoni. This does not bode well for my bank account balance. And the only way to *avoid* having such circular preferences is if there exists some "consistent preference ordering" of the three toppings - i.e. some ordering such that I will only pay to upgrade to a topping later in the order, never earlier. That ordering can then be specified as a utility function: a function which takes in a topping, and gives the topping's position in the preference order, so that I will only pay to upgrade to a topping with higher utility.

More advanced coherence theorems remove a lot of implicit assumptions (e.g. I could learn over time, and I might just face various implicit tradeoffs in the world rather than explicit offers to trade), and add more machinery (e.g. we can incorporate uncertainty and derive *expected* utility maximization and Bayesian updates). But they all require something-which-works-like-money.

Money has two key properties in this argument:

- Money is additive across decisions. If I pay $1 to upgrade anchovy to pepperoni, and another $1 to upgrade pepperoni to mushroom, then I have spent $1 + $1 = $2.
- All else equal, more money is good. If I spend $3 trading anchovy -> pepperoni -> mushroom -> anchovy, then I could have just stuck with anchovy from the start and had strictly more money, which would be better.

These are the conditions which make money a "measuring stick of utility": more money is better (all else equal), and money adds. (Indeed, these are also the key properties of a literal measuring stick: distances measured by the stick along a straight line add, and bigger numbers indicate more distance.)

Why does this matter?

There's a common misconception that *every* system can be interpreted as a utility maximizer, so coherence theorems don't say anything interesting. After all, we can always just pick some "utility function" which is maximized by whatever the system actually does. It's the measuring stick of utility which makes coherence theorems nontrivial: if I spend $3 trading anchovy -> pepperoni -> mushroom -> anchovy, then it implies that *either* (1) I don't have a utility function *over toppings* (though I could still have a utility function over some other silly thing, like e.g. my history of topping-upgrades), or (2) more money is not necessarily better, given the same toppings. Sure, there are ways for that system to "maximize a utility function", but it can't be a utility function over toppings which is measured by our chosen measuring stick.

Another way to put it: coherence theorems assume the existence of some resources (e.g. money), and talk about systems which are pareto optimal with respect to those resources - e.g. systems which "don't throw away money". Implicitly, we're assuming

that the system generally "wants" more resources (instrumentally, not necessarily as an end goal), and we derive the system's "preferences" over everything else (including things which are not resources) from that. The agent "prefers" X over Y if it expends resources to get from Y to X. If the agent reaches a world-state which it could have reached with strictly less resource expenditure in all possible worlds, then it's not an expected utility maximizer - it "threw away money" unnecessarily. We *assume* that the resources are a measuring stick of utility, and then ask whether the system maximizes any utility function over the given state-space measured by that measuring stick.

Ok, but what about utility functions which don't increase with resources?

As a general rule, we don't actually care about systems which are "utility maximizers" in some trivial sense, like the rock which "optimizes" for sitting around being a rock. These systems are not very useful to think of as optimizers. We care about things which steer some part of the world into a relatively small state-space.

To the extent that we buy instrumental convergence, using resources as a measuring stick is very sensible. There are various standard resources in our environment, like money or energy, which are instrumentally useful for a very wide variety of goals. We expect a very wide variety of optimizers to "want" those resources, in order to achieve their goals. Conversely, we intuitively expect that systems which throw away such resources will not be very effective at steering parts of the world into relatively small state-space. They will be limited to fewer bits of optimization than systems which use those same resources pareto optimally.

So there's an argument to be made that we don't particularly care about systems which "maximize utility" in some sense which isn't well measured by resources. That said, it's an intuitive, qualitative argument, not really a mathematical one. What would be required in order to make it into a formal argument, usable for practical quantification and engineering?

The Measuring Stick Problem

The main problem is: how do we recognize a "measuring stick of utility" in the wild, in situations where we don't already think of something as a resource? If somebody hands me a simulation of a world with some weird physics, what program can I run on that simulation to identify all the "resources" in it? And how does that notion of "resources" let me say useful, nontrivial things about the class of utility functions for which those resources are a measuring stick? These are the sorts of questions we need to answer if we want to use coherence theorems in a physically-reductive theory of agency.

If we could answer that question, in a way derivable from physics without any "agenty stuff" baked in a priori, then the coherence theorems would give us a nontrivial sense in which some physical systems do contain embedded agents, and other physical systems don't. It would, presumably, allow us to bound the number of bits-of-optimization which a system can bring to bear, with more-coherent-as-measured-by-the-measuring-stick systems able to apply more bits of optimization, all else equal.

Distributed Decisions

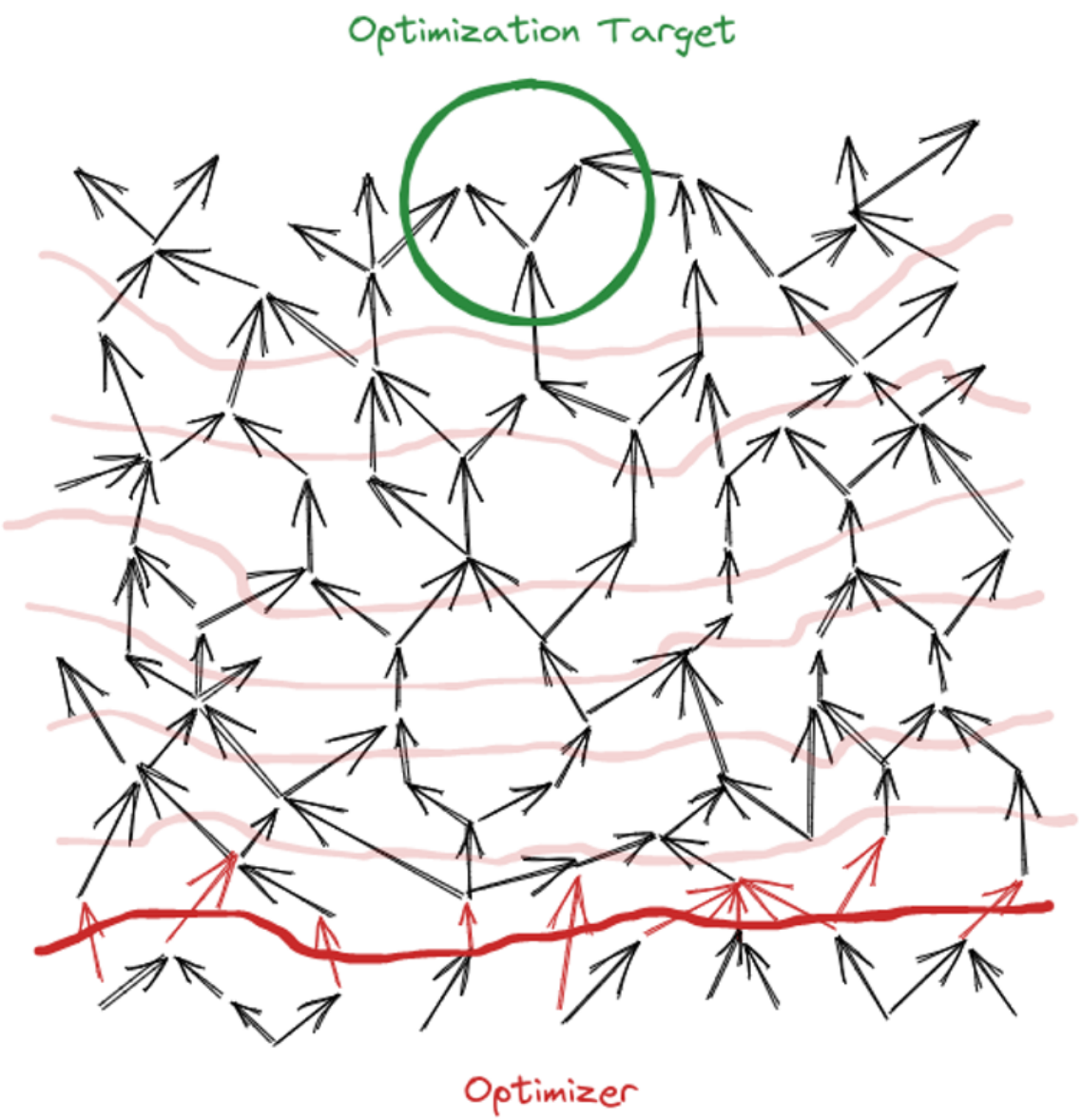
Crossposted from the AI Alignment Forum. May contain more technical jargon than usual.

Consider two prototypical “agents”: a human, and a company.

The human is relatively centralized and monolithic. As a rough approximation, every 100 ms or so observations flow into the brain from the eyes, ears, etc. This raw input data updates the brain’s world-model, and then decisions flow out, e.g. muscle movements. This is exactly the sort of “state-update model” which Against Time In Agent Models criticized: observations update one central internal state at each timestep, and all decisions are made based on that central state. It’s not even all that accurate a model for a human, but let’s set that aside for now and contrast it to a more obviously decentralized example.

In a company, knowledge and decisions are distributed. A cashier sees and hears customers in the store, and interacts with them in order to sell things. Meanwhile, a marketing editor tweaks some ad copy. Each mostly makes decisions based on their local information; most of that local information is not propagated to other decision-makers. Observations don’t update a single centralized state which informs all decisions. Instead, different decisions have different input information from different sources.

In Optimization at a Distance, I suggested a mental picture of agents kinda like this:



> Note that this particular drawing doesn't have any inputs to the optimizer (i.e. observations) for simplicity, but it's easy to add inputs. The optimizer need not be strictly causally upstream of the target; it could have interactions back-and-forth with the target.

It's like a phased array: there's lots of little actions distributed over space/time, all controlled in such a way that their influence can add up coherently and propagate over a long distance to optimize some far-away target. Optimization at a Distance mainly emphasized the "height" of this picture, i.e. the distance between optimizer and target. This post is instead about the "width": not only are the actions far from the optimization target, the actions themselves are also distributed in spacetime and potentially far apart from each other.

Contrast: Bayesian Updates

Suppose I want to watch my favorite movie, 10 Things I Hate About You, in the evening. To make this happen, I do some optimization - I steer myself-in-the-evening and my-immediate-

environment-in-the-evening into the relatively small set of states in which I'm watching the movie. Via the argument in Utility Maximization = Description Length Minimization, we should expect that I approximately-act-as-though I'm a Bayesian reasoner maximizing some expected utility over myself-in-the-evening and my-immediate-environment-in-the-evening. (Note that it's a utility function *over* myself-in-the-evening and my-immediate-environment-in-the-evening, not just any old random utility function; something like e.g. a rock would not be well-described by such a utility function.)

While arranging my evening, I may perform some Bayesian updates. Maybe I learn that the movie is not available on Netflix, so I ask a friend if they have a copy, then check Amazon when they don't. This process is reasonably well-characterized as me having a centralized model of the places I might find the movie, and then Bayes-updating that model each time I learn another place where I can/can't find it. (If I had checked Netflixed, then asked my friend, then checked Netflix again because I forgot whether it was on Netflix, that would not be well-modeled as Bayesian updates.)

By contrast, imagine that myself *and some friends* are arranging to watch 10 Things I Hate About You in the evening. I check to see if the movie is on Netflix, and at the same time my friend checks their parents' pile of DVDs. My friend doesn't find it in their parents' DVD pile, and doesn't know I already checked Netflix, so they *also* check Netflix. My friends and I, as a system, are *not* well-modeled as Bayesian updates to a single central knowledge-state; otherwise we wouldn't check Netflix twice. And yet, it's not obviously suboptimal (like me forgetting whether the movie is on Netflix would be). If there's a lag in communication between us, it may just be faster and easier for us to both check Netflix independently, and then both check other sources independently if the movie isn't there. We're acting independently to optimize the same goal; our actions are chosen "locally" on the basis of whatever information is available, not necessarily based on a single unified knowledge-state.

So, we don't really have "Bayesian updates" in the usual sense. And yet... we're still steering the world into a relatively narrow set of states, the argument in Utility Maximization = Description Length Minimization still applies just fine, and that argument is still an essentially Bayesian argument. It's still using a Bayesian distribution - i.e. a distribution which is ultimately part of a model, not necessarily a fundamental feature of the territory. It's still about maximizing expected utility under that distribution. My friends and I, as a system, are still well modeled as a "Bayesian agent" in some sense. Just... not a *monolithic* Bayesian agent. We're a *distributed* Bayesian agent, one in which different parts have different information.

Conditioning

Conditional probabilities do still enter the picture, just not as updates to a centralized world-state.

In the movie example, when I'm searching for the movie in various places, how do I steer the world into the state of us-watching-the-movie-in-the-evening? How do I maximize $E[u(X)]$, jointly with my friends? Well, I act on the information I have, plus my priors about e.g. what information my friends will have and how they will act. If I have information Y (e.g. I know that the movie isn't on Netflix, and know nothing else relevant other than priors) when making a particular decision, then I act to maximize $E[u(X)|Y]$.

Why that particular mathematical form? Well, our shared optimization objective $E[u(X)]$ is a sum over worlds $(X, Y, ...)$:

$$E[u(X)] = \sum_{X,Y,\ldots} P[X, Y, \ldots] u(X)$$

If I know that e.g. the movie is not on Netflix, then I know my current action won't impact any of the worlds where the movie *is* on Netflix. So I can ignore those worlds while making the current decision, and just sum over all the worlds in which the movie is *not* on Netflix. My new sum is $\sum_{X,\ldots} P[X, Y, \ldots]u(X)$, which becomes $E[u(X)|Y]$ after normalizing the probabilities. (Normalizing doesn't change the optimal action, so we can do that "for free".) By ignoring all the worlds I'm not in (based on the input information to the current decision), and taking the expectation over the rest, I'm effectively maximizing expected utility conditional on the information I have when making the decision.

More generally: **each action is chosen to maximize expected utility conditional on whatever information is available as an input to that action** (including priors about how the other actions will be taken). That's the defining feature of a *distributed* Bayesian agent.

This post (and the more dense version here) spells out the mathematical argument in a bit more detail, starting from coherence rather than utility-maximization-as-description-length-minimization.

(Side note: some decision theory scenarios attempt to mess with the "current action won't impact any of the other worlds" part, by making actions in one world impact other worlds. Something FDT-like would fix that, but that's out of scope for the current post.)

Resources

The "Measuring Stick of Utility" Problem talks about how grounding the idea of "resources" in non-agenty concepts is a major barrier to using coherence theorems to e.g. identify agents in a given system. If we have distributed decisions, optimization at a distance, or both, *and* we expect that information at a distance is mediated by relatively low-dimensional summaries (i.e. the Telephone Theorem), then there's an intuitively-natural way to recognize "resources" for purposes of coherence arguments.

Let's go back to the example of a company, in which individual employees make many low-level decisions in parallel. The information relevant to each decision is mostly local - e.g. a cashier at a retail store in upstate New York does not need to know the details of station 13 on the company's assembly line in Shenzhen. But there is some relevant information - for instance, if an extra 10 cents per item are spent at station 13 on the assembly line in Shenzhen, then the cashier needs to end up charging another ~10 cents per item to customers. Or, if the assembly line shuts down for a day and 10000 fewer items are produced, then the cashiers at all of the company's stores need to end up selling 10000 fewer items.

So we have this picture where lots of different decisions are made mostly-locally, but with some relatively small summary information passed around between local decision makers. That summary consists mainly of a sum of "resources'' gained/lost across each decision. In our example, the resources would be dollars spent/gained, and items created/sold.

The key here is that we have lots of local decisions, with relatively low-dimensional coupling between them. The summary-information through which the decisions couple is, roughly speaking, the "resources". (In practice, there will probably also be lots of extra summary-information between localities which isn't controllable via the actions, and therefore needn't be treated as a resource - e.g. all the facts about concrete one could learn from the store's walls which would carry over to the concrete in the factory's walls.)

Alternatively, rather than starting from distributed decisions, we could start from optimization at a distance. Because the optimization target is “far away” from the actions, only some relatively-low-dimensional summary of the actions impacts the target. Again, the components of that summary are, roughly speaking, the “resources”.

This picture fits in nicely with coherence theorems. The theorems talk about how a local decision maker needs to act in order to achieve pareto-optimal resource use, while still achieving local goals. For instance, the company’s marketing department should act-as-though it has a utility function over ads, otherwise it could run the same ads while spending pareto-fewer resources.

This picture also fits in nicely with natural abstractions. We have a large system with lots of parts “far away” from each other. The Telephone Theorem then says that they will indeed interact only via some relatively low-dimensional summary. In a decision framing, it says that only a relatively low-dimensional summary of the far-away decisions will be relevant to the local decision. Furthermore, we can in-principle derive that low-dimensional summary from the low-level physics of the world.

But this is still just an intuitive story. To make it rigorous, the Measuring Stick of Utility post argued that we need our resources to have two main properties:

- More resource is always better
- Resources are additive across decisions

Additivity across decisions, in particular, is the more restrictive condition mathematically. In order to identify natural abstraction summaries as “resources” for coherence purposes, those summaries need to be additive across all the local decisions.

… which is the main claim argued in Maxent and Abstractions. Summaries of information relevant at a distance can indeed be represented as sums over local variables/decisions.

What's General-Purpose Search, And Why Might We Expect To See It In Trained ML Systems?

Crossposted from the AI Alignment Forum. May contain more technical jargon than usual.

Benito has an interesting job. Here's some of the stuff he's had to do over the past couple years:

- build a prototype of an office
- resolve neighbor complaints at a party
- find housing for 13 people with 2 days notice
- figure out an invite list for 100+ people for an office
- deal with people emailing a funder trying to get him defunded
- set moderation policies for LessWrong
- write public explanations of grantmaking decisions
- organize weekly online zoom events
- ship books internationally by Christmas
- moderate online debates
- do April Fools' Jokes on Lesswrong
- figure out which of 100s of applicants to do trial hires with

Quite a wide variety!

Benito illustrates an interesting feature of humans: you can give humans pretty arbitrary goals, pretty arbitrary jobs to do, pretty arbitrary problems to solve, and they'll go figure out how to do it. It seems like humans have some sort of "general-purpose problem-solving" capability.

Now, there's more than one part of general-purpose problem solving. There's efficient information-gathering and model-building and updating. There's searching for promising plans. There's execution (or, in the organizational context, operations). A general-purpose problem-solver needs general-purpose versions of all those. But for this post, I want to focus on the "searching for promising plans" part.

First things first: what is this "search" thing, anyway?

Babble And Prune Is Not The Only Search Method

This whole post started out because I was talking about "search" (in the context of an inner alignment strategy) and it turned out that people had radically different pictures of what the word "search" means. In particular, it turned out that a bunch of people pessimistic about the strategy were picturing some variant of babble and prune: "babble" candidate solutions, then "prune" unpromising solutions, and hopefully iterate toward better and better solutions.

This is not really how humans search for promising plans. Consider, for example, a human planning a trip to the grocery store. Typical reasoning (mostly at the subconscious level) might involve steps like:

- There's a dozen different stores in different places, so I can probably find one nearby wherever I happen to be; I don't need to worry about picking a location early in the planning process.
- My calendar is tight, so I need to pick an open time. That restricts my options a lot, so I should worry about that early in the planning process.
    - <go look at calendar>
- Once I've picked an open time in my calendar, I should pick a grocery store nearby whatever I'm doing before/after that time.
- … Oh, but I also need to go home immediately after, to put any frozen things in the freezer. So I should pick a time when I'll be going home after, probably toward the end of the day.

Notice that this sort of reasoning mostly does *not* involve babbling and pruning entire plans. The human is thinking mostly at the level of constraints (and associated heuristics) which rule out broad swaths of plan-space. The calendar is a taut constraint, location is a slack constraint, so (heuristic) first find a convenient time and then pick whichever store is closest to wherever I'll be before/after. The reasoning only deals with a few abstract plan-features (i.e. time, place) and ignores lots of details (i.e. exact route, space in the car's trunk); more detail can be filled out later, so long as we've planned the "important" parts. And rather than "iterate" by looking at many plans, the search process mostly "iterates" by considering subproblems (like e.g. finding an open calendar slot) or adding lower-level constraints to a higher-level plan (like e.g. needing to get frozen goods home quickly).

So humans' general-purpose internal search mostly doesn't look like babble and prune in the classic sense. It's mostly operating on things like constraints and heuristics, abstraction, and subproblems. (There may be some babbling and pruning in there, but it's not doing most of the algorithmic efficiency work, and we're not directly babbling and pruning whole plans.)

In fact, even classic path search algorithms (like e.g. A* search) don't really resemble pure babble and prune on closer inspection. When using a classic path-search algorithm to e.g. find a route from LA to to Seattle, we don't actually babble lots of possible LA-Seattle routes and evaluate them. Rather, we come up with solutions to lots of *subproblems*: routes between LA and various possible intermediate points (like San Francisco or Sacramento or Portland). And in the case of A* search, we use heuristics (generated by constraint relaxation) to decide which subproblems to pay attention to.

So what is search, if not (as Ivan Vendrov recently put it) "enumerating possible actions and evaluating their consequences"? Well, I'd say that a "general-purpose search" process is something which:

- Takes in a problem or goal specification (from a fairly broad range of possible problems/goals)
- … and returns a plan which solves the problem or scores well on the goal

The "evaluation of consequences" thing is relevant, but it's not really about the internal operations of the search process. Rather, "evaluation of consequences" is the defining feature of whether a search process is doing its job correctly: to tell whether the search process is working, use the problem/goal specification to evaluate the

consequences of the plan generated, and see whether the plan solves the problem or scores well on the goal.

Note that “general-purpose search” still does not include all the pieces of general-purpose problem solving; there’s still things like gathering information, building and updating models, execution of plans, or recognizing when plans need to be updated. But it does include the planning part.

Also note that a general-purpose search process need not solve all possible problems, or even all compactly-specifiable problems; that would be NP-Hard (almost by definition). It does need to handle a broad range of problems, ideally in some realistic environment.

Retargetability and Recursive Structure of Search

One key feature of a general-purpose search process is that it’s *retargetable*: because it can take in any problem or goal specification from a fairly wide class, we can retarget it by passing in a different problem/goal. For example, a path-finding algorithm (like A*) can take any start and end point in a graph.

Retargetability plays well with the recursive structure of search. When finding a route from LA to Seattle, for instance, we look for routes from LA to Sacramento or San Francisco. Those subproblems are themselves search problems, and can themselves be solved with the same path-finding method: just pass LA and Sacramento as the start and end points, rather than LA and Seattle. More generally, with a retargetable search process, we can recursively call the *same* general-purpose search process with different inputs in order to handle subproblems.

Later on, when we talk about why one might expect general-purpose search to eventually show up in trained ML systems, that sort of recursion will be one big piece.

Heuristics and Generality

In practice, in order to search *efficiently*, we tend to need some kind of heuristics. In physical-world pathfinding, for instance, a useful heuristic is to try to reduce Euclidean distance to the goal. However, that heuristic isn’t useful for all problems; Euclidean distance isn’t very useful as an heuristic for e.g. moderating online debates.

If we need heuristics for efficient search, and heuristics are specialized, how is general-purpose problem-solving a thing? Do we need to learn a whole new set of heuristics every time our task changes at all? Obviously not, in practice; most of the things on Ben’s list were things he only did once, but he nonetheless did a decent job (I know this because I’m one of his users). But then how do we achieve generality while relying on heuristics? Two main paths:

- There exist general-purpose methods for generating heuristics
- Heuristics tend to depend on the environment, but not on the exact objective

General-Purpose Generators Of Heuristics Are A Thing

Probably the best-understood heuristic-generator is problem relaxation.

Here's how it works. Let's say I'm planning a trip to the grocery store. My planning problem has a whole bunch of constraints: I need to have enough time, I need to be at the right place, can't let the frozen goods melt, the car must have enough gas, etc, etc. As a first pass, I ignore (a.k.. "relax") most of those constraints, and only pay attention to a few - like e.g. having enough time. That narrows down my solution space a lot: there's only a few spaces in my calendar with enough time. I pick a time slot. Then, I start to add other constraints back in.

The "relaxed" problem, in which I only worry about the time constraint, acts as an heuristic for the full problem. It helps me narrow down the search space early on, and focus on plans which at least satisfy that one constraint. And because it only involves one constraint, it's relatively easy to solve the relaxed problem.

More generally, the steps of problem relaxation are roughly:

- Start with a problem with a bunch of constraints
- Relax (i.e. ignore) a bunch of the constraints
- Solve the relaxed problem
- Use the relaxed problem-solution as an heuristic for the full problem

Another example, this time from path search: if we're solving a maze, we can relax the problem by ignoring all the walls. Then the shortest path is easy: it's a straight line from the start to the end, and its length is Euclidean distance. We can then use Euclidean distance as an heuristic while exploring the maze: preferentially explore in directions which have shorter Euclidean distance to the end. In other words, preferentially explore directions for which the relaxed problem (i.e. ignoring all the walls) has the shortest solutions.

Problem relaxation probably isn't the only heuristic generation method, but it's relatively well-understood mathematically, and it's an existence proof. It shows that general-purpose generators of heuristics are a thing. We should expect to see such heuristic-generators used inside of general-purpose search processes, in order to achieve generality and efficiency simultaneously.

Heuristics Tend To Depend On The Environment, But Not On The Objective

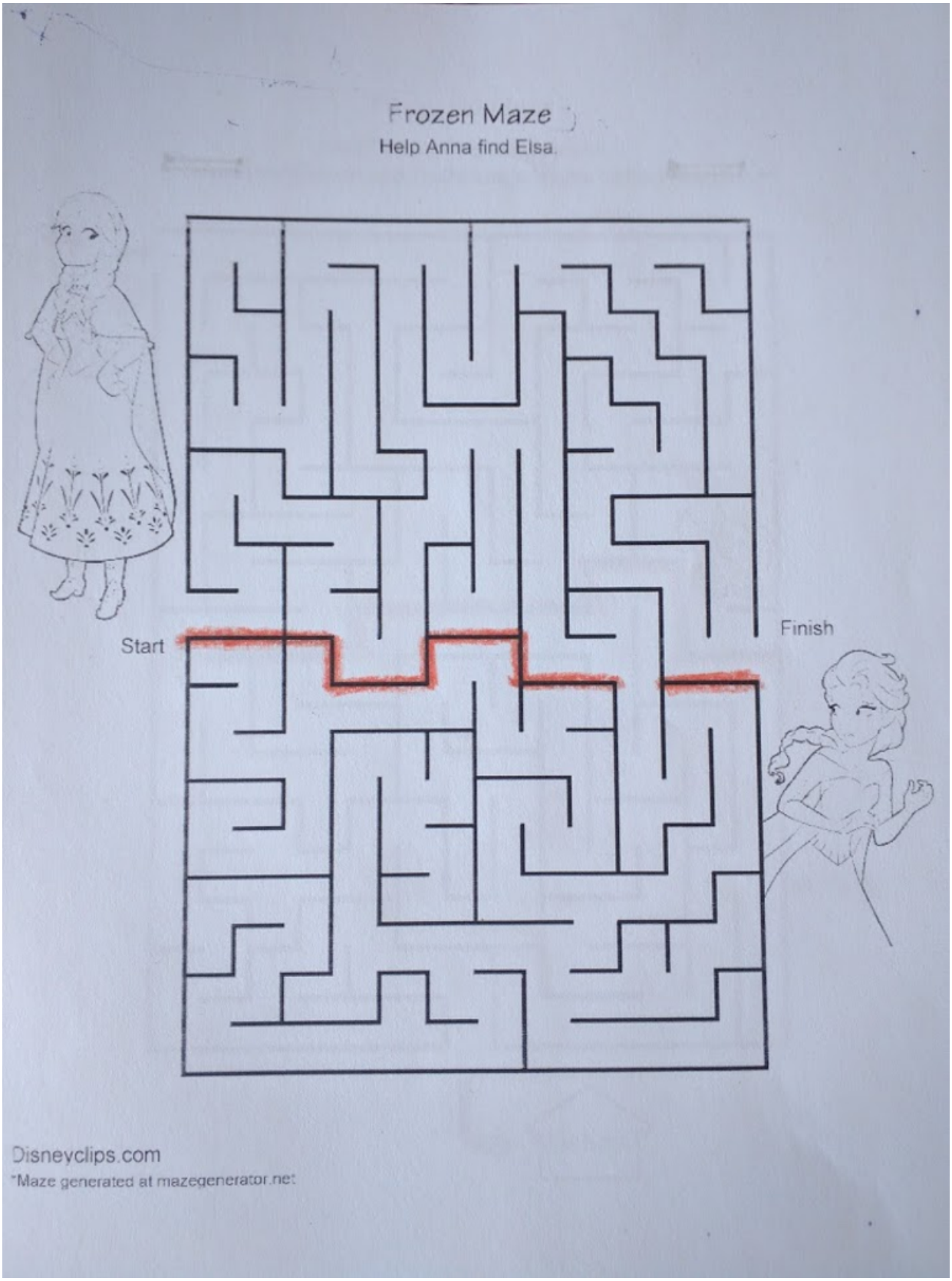
The previous section talked about two heuristics:

- When planning a grocery trip, pick a time slot while ignoring everything else
- When path-finding, prioritize shorter Euclidean distance to the end

Notice that both of these heuristics are very environment-dependent, but not very goal-dependent. If my time is very scarce, then picking a time slot first will be a good heuristic for planning all sorts of things in my day-to-day life, from meetings to vacations to workouts to a trip to the movies to some quiet reading. The heuristic does depend on "the environment" - i.e. it would be less useful for someone whose time is abundant - but it's relatively goal-agnostic.

Similarly, I can use Euclidean distance as a heuristic for finding road-trip routes between a wide variety of start and end locations. There are environments in which it's a terrible heuristic, but for road-trip planning it works decently for most start/end points.

Here’s a visual example which I love, an heuristic for path-finding in a maze from an old post:



Maze-specific, but it’s useful for a wide variety of start/end points.

The pattern: heuristics tend to be environment-dependent but relatively goal-agnostic.

Why would such a pattern apply?

One way to frame the answer: an environment typically includes a bunch of stable constraints, shared across problems in that environment. In our universe, the laws of physics are all constraints, human laws are all constraints, I can only move around so fast and carry so much, I only have so much time and money, my interfaces with other people/things only have certain knobs to turn, etc. And instrumental convergence means that it will very often be the same few constraints which are rate-limiting. So, insofar as we generate heuristics via constraint relaxation, we'll get environment-specific but reasonably goal-agnostic heuristics.

Another way to frame the answer: natural abstraction. Most far-apart chunks of the world only interact via relatively low-dimensional summaries; the rest of their interaction is wiped out by noise. So, optimizing those low-dimensional summaries will be a common heuristic across a wide range of goals within the same environment.

Side Note: Cached Solutions Are Not Heuristics (But Are Another General-Purpose Search Trick)

Another general-purpose search trick which someone will probably bring up if I don't mention it is caching solutions to common subproblems. I don't think of this as an heuristic; it mostly doesn't *steer* the search process, just speed it up. But it is a useful and common trick for improving efficiency. And we should expect it to speed up search over a wide variety of goals within the same environment, insofar as instrumental convergence applies in that environment. Instrumentally convergent subproblems come up over and over again for a wide variety of problems, so caching solutions to those subproblems is a general-purpose search accelerator.

Revisiting The Risks From Learned Optimization Arguments

We've now talked about how general-purpose search Is A Thing. We've talked about how general-purpose heuristics (and other general-purpose search tricks, like caching) Are A Thing. And we've observed that these things show up in humans. But why do they show up in humans? And should we expect them to show up in ML, or in intelligent aliens, or in other evolved/trained/selected agenty systems?

Key Idea: Compression Is Favored By Default

In general, evolved/trained/selected systems favor more compact policies/models/heuristics/algorithms/etc. In ML, for instance, the fewer parameters needed to implement the policy, the more parameters are free to vary, and therefore the more parameter-space-volume the policy takes up and the more likely it is to be found. (This is also the main argument for why overparameterized ML systems are able to generalize at all.)

The outer training loop doesn't just select for high reward, it also implicitly selects for compactness. We expect it to find, not just policies which achieve high reward, but policies which are very compactly represented.

At the same time, assuming the system encounters a wide variety of problems in its training environment, it needs generality in order to perform well. But compactness means that, when possible, it will favor generality using a smaller number of more general pieces rather than a larger number of more specialized pieces.

So things like general-purpose heuristics, general-purpose heuristic generators, and general-purpose search are exactly the sort of things these systems should favor (assuming the architecture is expressive enough, and the environment varies enough).

That's basically the argument for inner agents from Risks From Learned Optimization:

> In some tasks, good performance requires a very complex policy. At the same time, base optimizers are generally biased in favor of selecting learned algorithms with lower complexity. Thus, all else being equal, the base optimizer will generally be incentivized to look for a highly compressed policy.
>
> One way to find a compressed policy is to search for one that is able to use general features of the task structure to produce good behavior, rather than simply memorizing the correct output for each input. A mesa-optimizer is an example of such a policy. From the perspective of the base optimizer, a mesa-optimizer is a highly-compressed version of whatever policy it ends up implementing: instead of explicitly encoding the details of that policy in the learned algorithm, the base optimizer simply needs to encode how to search for such a policy. Furthermore, if a mesa-optimizer can determine the important features of its environment at runtime, it does not need to be given as much prior information as to what those important features are, and can thus be much simpler.

On top of all that, the recursive nature of search - the fact that recursively searching on subproblems is useful as a search technique - favors a retargetable general-purpose search process, as opposed to a hardcoded optimizer.

Takeaways

It seems like a lot of people have a picture of "search" as babble and prune. They correctly notice how inefficient babble and prune usually is, and conclude that it probably won't be selected for in trained ML systems.

But there's a more general notion of general-purpose search. It's a process which takes in any problem or goal from a wide class, and returns a plan which solves the problem or achieves the goal. That's the sort of search we expect to see show up in trained systems. The key property is that it's *retargetable*: the search process can take in a wide variety of goals. That retargetability is selected for due to the recursive structure of search: recursively running search on subproblems is a widely-useful technique, and that technique can be encoded much more compactly with a retargetable search process on hand.

In order for that search to run efficiently, while still maintaining generality and compactness, it will probably need to either generate heuristics in a general way, or leverage goal-agnostic (but possibly environment-specific) heuristics. Both of those are plausible options: problem relaxation is an existence proof of general-purpose generators of heuristics, and the shared constraints of an environment (plus natural abstraction and/or instrumental convergence) offer a source of goal-agnostic heuristics.

Some strategic upshots of all this:

- Possibility of rapid general capability gain if/when a system groks general-purpose search, especially if many general-purpose heuristics are learned first
- Retargeting the search process as an (inner) alignment strategy
- Relatively high chance of agent-like internal structure, like e.g. a reusable general-purpose search module which takes in an explicit objective