Has been posted to LessWrong: https://www.lesswrong.com/posts/xERh9dkBkHLHp7Lg6/how-stvs-may-make-it-harder-for-an-agi-to-trick-us

Summary / Preamble

I hope and think this article reads well in isolation, even though it is part of a series.

AI Alignment could be seen as an area of inquiry with many sub-areas. The area I focus on here is ways we might use a superintelligent AGI-system to help with creating an aligned AGI-system, even if the AGI we start out with isn't fully aligned.

Imagine a superintelligence that "pretends" to be aligned. Such an AI may give output that *seems* to us like what we want. But for some types of requests, it's very hard (even for a superintelligence) to give output that *seems* to us like what we want without it *actually* being what we want. Can we obtain new capabilities by making such requests, in such a way that the scope of things we can ask for in a safe way (without being "tricked" or manipulated) is increased? And if so, is it possible to eventually end up with an aligned AGI-system?

One reason for exploring such strategies is contingency planning (what if we haven't solved alignment by the time the first superintelligent AGI-system arrives?). Another reason is that additional layers of assurance could be beneficial (even if we *think* we have solved alignment, are there ways we could relatively quickly add additional layers of alignment-assurance?).

When dealing with a genie/oracle, we may not want to ask it to provide some direct solution/answer to what you want (and have it argue why it's a good idea). Instead we obtain what we want more indirectly, by having different instances of the genie/oracle aid us in the construction of more narrow/specialized programs (that score high in terms of verifiability).

We could call such programs STVs, where STV is an abbreviation for ***S****pecialized*

***T**ransparent **V**erifiable Program*. “STV” is a blurry concept (much like how “AI” and “AGI” are blurry concepts). An STV could be (but would not have to be) a narrow AI. More colloquially, STVs could also be referred to as “generators”.

In this text I explore principles/techniques/strategies for using STVs in ways that enable us to get things we want from an AGI without being “tricked”.

Future parts of this series will include discussion of:

- AI-generated computational proofs/argument-trees, and methods/strategies for verifying such proofs/arguments.
- Ideas for how formalisms for computational proofs could try to incorporate human cluster-like concepts, concept vagueness, formalism-to-reality mappings, references to code and results from running code, etc (from within themselves).
- How different “pieces” outlined in this series might be put together. Can we get from unaligned AGI to aligned AGI without being "tricked" along the way, and without being stopped by chicken-or-egg problems?
- Ideas for how we might want to make a system of sub-systems consisting of various "siloed" AGI-systems (with different settings, architectures, and alignment methodologies).
- Strategic and moral considerations, and possible first steps when starting to use superintelligent AGI-systems to do things outside of the digital realm.

Are any of the ideas in this series new? See here for a discussion of that.

Me: I think it may be helpful to start out with a simplistic example.

Me: So let’s imagine that we are interested in finding good chess-moves, and want an AGI to help us with this.

Imaginary friend: Ok.

Me: We *could* ask it directly for good chess-moves, along with explanations for why these chess-moves seem like good chess-moves.

Me: But if we want to make it hard for the AGI to “trick” us (to pretend that it’s giving us what we want, but not actually doing so). One way could be to ask for a *program* that searches for good chess-moves (instead of asking directly for the moves themselves).

Me: This could be beneficial for several reasons:

1. If the AGI wants to trick you in specific ways for specific strategic reasons, this may be harder when you force it to answer in a general way (not answer in a way that is custom to the specific thing you want).

2. If the AGI has strategic reasons for wanting to trick you about specific things, this is made harder when you don’t ask it specifically for what you are after (thus making it less straight-forward to guess what the answer will be used for).
3. Code that is specialized for some narrow/constrained task can often be easier to understand. This presumes that the AGI chooses to write the code easy to understand for humans, but if we know it is superintelligent and it’s *pretending* to be aligned then there are limits to what it can get away with (we would notice if the code clearly isn’t optimized for human understandability).
4. It may be easier for humans to understand chess programs than analyzing chess-moves directly (presuming that the program is written in a way that favors human understanding).
5. Narrow/specialized programs may more often be possible to code in a way where properties of behavior/output can be “proven” using formal/computational proofs. To put it simplistically: The more narrow a program is, the more suspicious it would be if the AGI can’t “prove” that it does what we want it to do.
6. Programs that solve problems in a general way - but within a narrow/constrained domain of tasks/questions - can more easily be tested against examples and real-world data. In the case of a chess-program we could for example check if it beats other chess-programs while using less computation.

Imaginary friend: Ok, but being able to confirm that a chess program is good at chess doesn’t solve the alignment problem. We want to be able to use the AGI to deal with the real world. And the real world is messy and complex in ways that board games aren’t, making verification and formal “proofs” much harder.

Me: Sure, I agree with that. I started out with a simplistic example, so as to be able to outline some principles without getting bogged down in details. And more principles and ideas will be conveyed later on. How far this gets us remains to be discussed.

Clarifying what I mean by STVs (aka “generators”)

Me: The term STV is a term I made up. Maybe other people have referred to similar concepts using different terms.

Me: It stands for *“**S**pecialized **T**ransparent **V**erifiable Program”*.

Me: By <u>*specialized*</u> I mean that the range of questions/tasks it is designed to handle is constrained. For example, maybe it only does one type of task, and only handles input from a specific domain and in a specific format.

Me: In some cases the program may work well without anything that resembles reasoning. In other cases, reasoning-like processes may be necessary. In such cases the reasoning should

(insofar as feasible) be constrained, specialized/non-general, transparent/interpretable, and closed off from other components/subroutines.

Me: By ***transparent*** I mean that it's easy for humans to understand how it works. Maybe not easy in an absolute sense, but no harder than necessary.

Me: Insofar as feasible the code should be modular, and written in a way that is optimized for making it easy for humans to understand (and hard for humans to misunderstand).

Me: There should be additional tools for analyzing the code beyond looking at source code. Diagrams, high-level textual descriptions, and maybe interactive overviews of various kinds.

Me: If the program needs components that are more "black-box-like" (like neural networks sometimes can be), then these components should be constrained (in terms of how they can influence the rest of the system, and in terms of what they are used for).

Me: By *verifiable* I mean that the behavior of the code can be verified. And much like the other properties of STVs, the property of being *verifiable* is not binary/either-or. It's more of a scale.

Me: One example of verification is formal/computational proofs. The best case is if good computational proofs can show that the program as a whole does what it's supposed to do. If that's not possible, proofs may still help with other things. Such as proving the behavior of subroutines. Or proving properties that the program reliably will approximate by some margin.

Me: Sometimes it may be possible to construct proofs about whether or not it's possible to prove that the system does what we want.

Me: Another example of verifiability is tests that should be satisfied by functions/modules. Concepts such as test-driven development are well-established among humans, and a superintelligent AGI-system could be expected to write much more comprehensive tests than a human.

Me: And another important example of verification is to test the system as a whole against examples and/or real-world data. For example:

- If the STV approximates physics somehow, then it could be compared against real-word data.
- If the STV predicts some aspects of human behavior, then it could be tested against actual data from actual humans (which the AGI-system that created the STV didn't have access to).
- If the STV generates proofs of some kind, then we can test if it is able to reliably find proofs that already exist (but which the AGI-system that created it didn't necessarily have access to).

“Hidden” behaviors in STVs

Imaginary friend: An unfriendly AI may create STVs in such a way that they pass our verification-process, but still have unfriendly behaviors “hidden” inside them.

Imaginary friend: Such “hidden” behaviors may be subtle and hard to find. They can’t necessarily be “pointed to” in some specific lines of code.

Imaginary friend: Do you imagine that we can guarantee against such “hidden” behaviors?

Me: I agree that this is an important concern. The degree to which we can guarantee against this - or at least make it unlikely/implausible - is from my perspective an open question (that may be worthy of further thinking/analysis).

Me: But this text touches upon several techniques and principles that I think can make it harder for an AGI-system to “hide” things.

Me: One way to make it harder to “hide” misbehavior is to look for ways that the STV can be made more general. To see why that potentially could be helpful, you could imagine a calculator. A calculator is specialized in the sense that it only does arithmetic. But if a calculator has special-purpose circuitry for handling 237+418 - or special-purpose functionality for handling multiplication using numbers that are within some specific range - well, that would be suspicious.. I’m giving a simplistic example here, but the same principle should also apply to more complex cases.

Me: So we could have STVs that look for ways other STVs can be made more general/simple.

Me: Another way to make it harder to “hide” misbehavior is to not let the AGI itself do the implementation of the sub-modules, but instead make other STVs do the implementation based on high-level specifications.

Me: Also, it may in some cases make sense to have STVs that generate other STVs. That is to say, if an STV can be described as an “answer-generator”, then an STV that outputs other STVs could be described as an “answer-generator-generator”. This is an even more indirect way of obtaining what we want from an AGI, and I’ll give some concrete examples of this later.

Me: Another thing that makes it harder to “hide” things is if the same functionality is to be implemented in different ways, based on varying principles/architectures (since it’s more challenging to make different implementations fail in the same way - even more so if it’s not the same AGI that makes both).

Why-not-bothing & output-convergence

Me: As a general principle, the more reasons we have to believe something, the more confident we can feel.

Me: For example, mathematicians may feel more confident in a theorem if it has been proved in several different ways (with varying proof-methods and starting assumptions).

Me: If there are 2 different alignment-methodologies that seem promising, then maybe we would want to use both as oracles/genies side by side, and see if they give the same answers. And if there are 3 different alignment-methodologies that seem promising, or 4 - well, you get the picture..

Me: At almost every level of a system - from the top level to detail-oriented sub-modules - it may often be helpful to try different architectures/implementations/principles, and see if the output is the same.

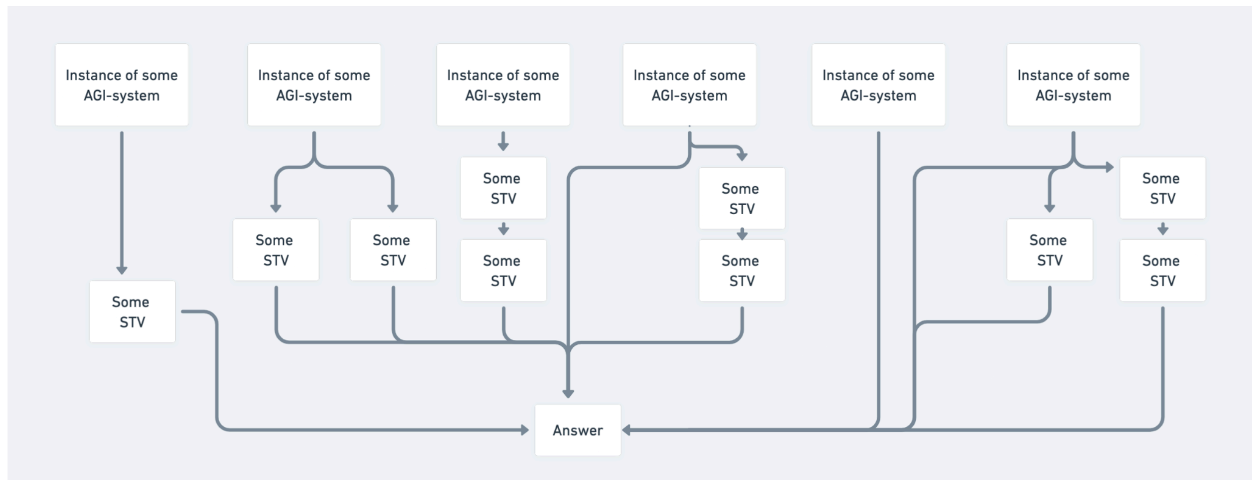
Me: We might sometimes refer to this as the principle of *"why not both?"*:

Me: Here are some examples (not a complete list):

- If we tell an AGI to make an STV for some task/purpose, then the AGI may come up with various different principles and architectures that such an STV could be based on. In such cases, we may want all the STVs, so that we may use them as one (running them side by side, and seeing it as suspicious if they don't produce converging output).
- Should AGI-systems have access to the history of requests it has been given earlier, or should it be reset before new requests are asked? I think the second often is more advisable, but sometimes it may make sense to try both.

- Should we give AGIs lots of background-knowledge that may be helpful for the task (but also may be helpful for tricking or manipulating us)? Or should we give it as little information as possible? Or maybe something in between? A good answer may often be yes, yes and yes (we can run different instances of the same AGI-instance).
- Who should make any given STV? And for the sub-modules in the architecture of some STV, who should make those? Often it is a good idea to have several implementations from several systems (with varying degrees of human involvement). And then we can see if outputs converge (or if one implementation can be proven to behave the same as another in a "mathematical proof"-like way).
- When we wonder about something, do we want to ask the AGI directly, or do we want an "answer-generator", or maybe an "answer-generator-generator"? Sometimes the answer may be *"yes, yes and yes"*.

Me: This kind of thing adds complexity, but with the help of a capable AGI-system it should be possible to set up systems that can help us handle such complexities.



Me: Another recurring theme in this text is indirection and subroutine reuse. If we let the AGI answer directly, and build whole systems by itself in one swoop, then it is for various reasons easier to "hide" misbehavior in the output. But when the AGI outputs smaller "pieces" of functionality that do some specialized thing, and can be put together in various ways to do various different things, then this can give more opportunities to notice if some of the "pieces" don't always do what they're supposed to do.

Me: If an AGI is asked to output systems that are optimized for verifiability, and the systems it suggests don't exploit these principles, then this may be a good cause for suspicion!

Me: The space of all claims and inference-steps that are true is huge, and it's deeply interconnected. And the more of this network you need to account for, and the more "nodes" in the network that can be verified, the harder it is to get away with false claims without contradicting yourself (especially when the subset of the network you account for is dense).

More comprehensive and systematic ways of leveraging this principle is one of the things that will be explored in part 3 of this series.

Human-emulating STVs

Me: One thing STVs maybe could be made to do is to predict human responses. Like, predicting what a human would think of some argument, how a human would evaluate some piece of code, etc.

Imaginary friend: Aren't STVs supposed to be "narrow" though? Humans are in a sense AGIs.

Me: We are indeed general reasoners. And this makes it harder for STVs to predict humans (while remaining transparent/verifiable), compared with an STV that is less in need of general reasoning.

Imaginary friend: But you still think that we - with the help of an AGI - could obtain STVs that predict human thinking/behavior? And that we to a sufficient degree could verify that such STVs actually do what we want them to?

Me: It seems likely to me that we could. But it also seems plausible that we wouldn't be able to.

Me: Keep in mind:

- There are degrees of success. For example, sometimes we may be only 90% confident that an STV works as it should. In such cases, whether we should use it depends a lot on context/specifics. If it is a component in a larger system, then there may be ways to use it where it only can help (in certain instances, if it works), and doesn't have much opportunity to do damage.
- Human-emulating STVs would not need to always have an answer. For example, if an STV has the job of predicting how a human would categorize something, it could be ok for it to sometimes not know the answer.

Imaginary friend: How would an STV predict human behavior though?

Me: Here are some ideas:

Way for STV to predict human behavior	Details
Brain emulations	It's sometimes pointed out that human brain emulations could have large advantages in terms of alignment, but that the first AGIs are unlikely to be

	brain emulations. But might we have a potentially unaligned AGI help us obtain brain emulations in a safe+verifiable way, without letting it “leave the digital realm”? The outputs of brain emulations can be tested against real-world data (data from brain scanning and so on), and the emulations can also be tested for how good they are at predicting the actions/answers/etc of humans (and other animals). STVs that emulate brains need not be given to us directly from an AGI. There would be various other options (that aren’t mutually exclusive). Could it, for example, be possible to generate an STV that generates brain emulations based on the DNA of the animal in question (but without being given any direct info about brains)? Might the functionality of such an STV be made in a very general-purpose way (where it’s hard to “hide” details)? Might the functionality used to indirectly generate brain emulations also be used to generate other things, which may be verified (predictions regarding the inner workings of the gut, simulations of the details of the inner workings of a flower, etc)? One dilemma in regards to simulations is how fine-grained they are, and how they handle a model where the details are unclear (they simulate something that exists in the real world, but they are not given precise and accurate data of starting conditions). This is not just a dilemma for brain simulations, but for simulations of any physical system. Something we may want is a system that gives an accurate description of the *range* of possible outcomes, given a description of the *range* of possible starting conditions. And we want the possibility for the simulation to not spend lots of computation on details we don’t care about (only computing details that are useful, or that are helpful for verification of simulation). Since these are general-purpose challenges, which aren’t specific to brain simulations, we may want to have STVs that can help generate “simulation-approximations” for any physical system. That way we can *also* test if they do a consistently accurate job when used to make predictions about other physical systems (and not *only* be tested for whether or not they do a good job with brain emulations).

	When doing anything that resembles emulating a brain, it is very important to avoid/minimize risk of suffering subroutines! Failing at this could result in mind crimes and suffering, potentially at an enormous scale! At every step of the process we should: 1. Avoid simulations that might be conscious. 2. Avoid simulating processes that would be likely to experience significant suffering if we were wrong about #1. Subroutines with positive valence may often be unproblematic, or even a good thing. But it remains to be seen how good our understanding of consciousness will become (the consequences of assuming wrongly can be very bad!).
Approximations of brain emulations (sometimes based on indirect methods)	In a sense any brain emulation can be seen as an approximation of a more high-fidelity emulation, but what I mean here is that large components of the STV need not be based on “emulation” at all, as long as the STV predicts aspects of brain states + what the human answers/does. In a sense, knowing what the human says/does may be all we are interested in, but if it makes predictions about brain states then this may make verification easier (especially if the STV is based in part on assumptions about which brain states follow from which, which actions correspond to which brain states, brain state sequences that cannot happen, etc).
Lots of raw and somewhat “hand-coded” probabilistic and modular inference-rules that encode typical human responses within some domain/context	Inference-rules could reference results from subroutines that use neural nets, but if so we should probably require that we can verify what aspects of the thinking/work is done by the neural net. The “core” of the inference should be done in a more symbolic/interpretable way. Maybe bayesian networks, or something similar, could be part of the system somehow. Imagine if a smart/cooperative human sits in a room, and is given simple multiplication-questions. I would guess that in such a situation we would not need high-fidelity brain-emulation to predict the humans “output” (a calculator could suffice!). This simplistic example could work as a weak “existence-proof” of sorts, showing that in restricted situations/domains,

	the most probable human output can be predicted without using brain emulation. But whether this can be done in a verifiable way for useful tasks is AFAIK an open question. It's sometimes pointed out that it's infeasible to "hand-code" what we mean by various fuzzy concepts (such as "person", "human", "animal", "dead", "happy", etc). But even if that's infeasible for *us*, it's not necessarily infeasible for a superintelligence. And if a superintelligence hand-codes it, there may be ways of verifying that the hand-coded specification does a good job of mimicking human output. The AGI would not necessarily do the "hand-coding" directly itself. It could output STVs that do the "hand-coding" (based on various methods). Some such STVs might do the hand-coding based on being given books and internet archives, and building a model of human concepts from what it reads/hears/sees.

Me: If we end up in a situation where an AGI can make STVs for us that predict human behavior, we might wish that we had more experimental data to test those STVs against. That's one example of why it may be useful to plan in advance!

Me: For example, if we think we might want to use human-mimicking STVs to evaluate proofs/arguments provided by AGIs/STVs, but in a piecemeal fashion, then it might be helpful to think ahead of time about what the smallest components/steps of such proofs/arguments (that can be evaluated in isolation) might look like.

Me: And if we want STVs that for example mimic humans looking over code, then that is also something that may be helpful to plan for in some detail.

Me: Some STVs may be easier to verify if we have brain state data, e.g. from MRIs, of humans that do the exact same types of tasks that the STVs emulate humans doing. Sometimes the STVs may emulate people sitting in a similar room as in the experiment, in front of a similar computer to the one in the experiment, etc.

Me: STVs should be able to describe patterns about how various brain states correspond to both actions (answers, code that is written, etc) and other measurements (posture, eye movement, milliseconds between keystrokes, mouse movement, brain state measurements, etc). Preferably these patterns should be as general as possible (e.g. not just for people with red hair sitting in rooms with yellow pain when the room is 35°C).

Me: The more experiments we have, and the data we have from experiments (mouse movement, eye movement, video of posture, brain measurements, etc), the more challenging it may be for an STV to “make things up” (without this being discovered when predictions are tested against existing data).

Me: It may also be helpful to have additional data about humans who participate in experiments (with the informed consent of participants, of course). Their DNA, bodily features, their gut microbiome, etc, etc.

Me: Often it’s not the average human that we want STVs to predict, but rather humans who are usually high in intelligence and [cognitive reflection]() (and are talented at what they do).

STVs that help with software-development

Me: Another thing STVs could help with is software development. Here are some examples:

What	Details
Rewrite code in ways that are proved to not change behavior	It may be relatively tractable to prove that two pieces of code behave similarly and always will have the same output (and to verify such proofs). If you are a programmer, you know ways to predict that a code-modification won't change output. For example, you know that an if-else statement could be replaced by a switch-statement, or that a+b can be replaced with b+a when a and b are numbers. Sometimes you make mistakes when doing this type of reasoning, but this doesn’t mean that proofs that use similar reasoning are impossible (you sometimes make mistakes when doing math as well, but that doesn’t make it impossible to construct mathematical proofs!). These kinds of proofs could be computational, meaning that to mechanically check the proofs would be relatively trivial. And all that is needed to show that a given proof is wrong is one counter-example (2 pieces of code that are “proven” to have the same output/behavior, but have different output/behavior when we run them with some specific input). Such a counter-example would not only invalidate that specific proof - it would be a cause for questioning the proof-system itself (and whoever made it). The better and more extensively a proof-system has been tested, the better.

<table>
<tr>
<td></td>
<td>Reasons for rewriting code, and proving equivalence between pieces of code, could include:

Rewriting code to be more computationally efficient
Rewriting code so as to score higher in terms of how easy it is for humans to understand it
Using these kinds of proofs as building-blocks in other proofs
</td>
</tr>
<tr>
<td>Use code-rewrites with proofs as building-blocks in other proofs</td>
<td>One example could be proofs showing specifically and precisely how the behavior of two pieces of code are different. Another example could be proofs showing how one piece of code approximates another piece of code.

Here are some examples where I try to allude to what I have in mind (handwavy, but still dense, so feel free to skip):

“Function A has equivalent output to function B, except for when the input is in range C, in which the output of A is equivalent to the output of Function D when D is given output from B as input”
“Using search-procedure A we can’t find any computationally efficient way of choosing input for Function A such that the output from A doesn’t approximate the output of Function B, with the probability-distribution that describes this approximation being within probability-distribution-space C”
“Among the space of allowable input for function A, there is no sub-space of size larger than B where the output will approximate function B according to approximation-description C, with the exception of input-spaces for which function A always will return an error”
</td>
</tr>
<tr>
<td>Convert between code and high-level descriptions/specifications of code (and look for discrepancies)</td>
<td>What a piece of code is supposed to do can be described at various levels of specificity, and in various ways:

You can describe what it does with text
You can make diagrams
You can give specific examples of what the output is (or should be) given specific input
You can list things that should be true about the output (either all the time, or presuming certain things being true about the input)
You can have various interactive tools that let you explore the specification of what a piece of code does / is supposed to do. Letting you see step by step how various inputs would be processed, and
</td>
</tr>
</table>

	letting you "zoom" in and out of modules/subroutines. When you have STVs that can convert between code and more high-level descriptions of code, one thing you can do is to search for code with significantly different functionality/behavior that would generate the same high-level description. This could potentially be helpful in finding out how some description is underspecified in ways that matter. STVs like this could be used to make it harder for other systems (STVs, AGIs, etc) to "hide" details in code that they write - by not letting them implement all the details themselves There need not be only one STV that implements a high-level description. Different STVs could implement the code, and the code they output could be compared (by testing if output converges, and by making proofs about how the code-bases are equal/similar/different in terms of behavior). We could make "loops" where we convert from high-level descriptions, to code, to high-level descriptions, to code, etc - with each step being done by different/independent STVs. And then we can see if we get "drift", or if we get more or less equivalent code-bases and high-level descriptions again and again.
Look for bugs in code	Some bugs are bugs in a very unambiguous sense, making them clear targets. Other bugs are more subtle and dependent on nuanced human-like judgements. I wouldn't expect some very non-AGI-like SVG (that understands code and nothing else) to robustly be able to find all bugs. But probabilistically it could help. Having SVGs based on various principles/architectures/settings that search for bugs could make it much harder to "hide" bugs.
Help to write/propose tests	When specifying what code should do, it may be helpful to write tests that can detect examples of code doing what it should do and examples of code not doing what it should do. This can be done by humans, but STVs may also suggest tests.
Look for structural similarities/overlap between different code bases	These STVs could search for ways to rewrite the different code bases so as to minimize the amount/size of non-overlapping modules.

	A somewhat similar task would be to give the STV access to existing code-bases/libraries while it codes, and have it try to maximize overlap with these existing code-bases/libraries (as one of the things it maximizes for). Another somewhat similar task would be to be given a code base that does one thing, and search for the minimal amount of changes that could be made so as to make that code-base do another thing. (For example, the minimal amount of changes to convert a narrow AI that does one thing into a narrow AI that does some other thing.) Overlapping modules can more easily be tested in an extensive way. Especially if the STVs that rely on the modules vary in terms of how easy it is to verify if they do a good job or not.
Identify parts/aspect of code that don't only significantly affects output small sections of the space of possible inputs	Are there ways to rewrite code that would not change output for the great majority of inputs, but would change output for a few inputs? Are there ways to simplify code that *would* change output for most input, but would *not* significantly change the average score of the output (as ascribed when it's evaluated)? If efficient ways can be found of identifying such changes (when they exist in the space of possible changes), then maybe this could be used to make it harder to intentionally "hide" malicious behavior in STVs. If we call the part of input-space where output can be verified A, and the part of input-space where we can't verify output B, are there ways to rewrite the code (without making the code-base more extensive) that wouldn't affect output within A negatively, but would have a significant effect on some of the output within B? If so, that may often be a reason to have less trust in that STV (and the system that produced it). Especially if the modified STV and the non-modified STV could be made to produce contradicting output.

Scope of what STVs could be used for (without becoming too AGI-like)

Imaginary friend: If it is hard to verify that an AGI does what you want, then presumably the same will be the case for STVs the more similar they are to an AGI?

Imaginary friend: Many tasks seem to be more or less AGI-complete, which presumably means that it would be hard for an STV to do them without being AGI-like.

Imaginary friend: How broadly capable do you think an STV can be while still being relatively *narrow/specialized*, and scoring well in terms of *transparency* and *verifiability*?

Me: I don't have any arguments that are watertight enough to justify opinions on this that are confident and precise. But my gut feeling is somewhat optimistic.

Imaginary friend: One thing you should keep in mind is the limits of symbolic reasoning. Earlier in the history of AI, people tried to make expert systems that rely heavily on "neat" symbolic reasoning. But these systems were largely unable to deal with the complexity and nuance of the real world.

Me: But there is a huge difference between what an unassisted human can make and what a superintelligent AGI should be able to make. If a superintelligent AGI doesn't do a much better job than a human would at coding systems that are transparent and verifiable - well, that would be suspicious..

Me: Yes, people have worked on systems that rely heavily on explicit reasoning. But everything that *has* been tried is very crude compared to what *could* be tried. A superintelligent AGI would presumably be much less limited in terms of what it would be able to achieve with such systems. More creative, and more able to create sophisticated systems with huge amounts of "hand-coded" functionality.

Me: There is this mistake many people have a tendency to make, where they underestimate how far we can get on a problem by pointing to how intractable it is to solve in crude/uncreative ways. One example of this mistake, as I see it, is to say that it **certainly** is impossible to prove how to play perfect chess, since this is impossible to calculate in a straight-forward combinatorial way. Another example would be to say that we cannot solve protein folding, since it is computationally intractable (this used to be a common opinion, but it isn't anymore).

Me: Both in terms of being "optimistic" and "pessimistic", we should try to avoid taking for granted more than what we have a basis for taking for granted. And this also applies to the question of *"how well can a program reason while constrained by requirements for verifiability/provability/transparency?"*.

Me: One way to think of intelligence is as doing efficient search in possibility-space. To put it a bit simplistically:

- A board game AI searches for strategies/moves that increase the probability of victory.
- A comedian searches for things to say and do that make people entertained in a comedic way.
- A programmer searches for lines of code that results in programs that score high in terms of certain criteria.
- An inventor searches for construction-steps where the construction-steps and the resulting physical systems scores high in terms of certain criteria.
- A digital hacker searches for ways to interact with digital systems that result in behavior/outcomes that the hacker wants (contrary to wishes of designers of said digital systems).
- A theorem prover searches for proof-steps that can prove whatever it's trying to prove.
- Etc, etc

Me: Another way to think of intelligence is as being able to build an accurate and extensive model of some domain. Having an extensive model of the domain sort of implies often being able to answer questions of the form *"which options/strategies/possibilities rank high given conditions x and preferences y?"*. Which implies being good at efficient search through possibility-space.

Me: In order to be good at searching through possibility-space efficiently, here are some capabilities that may help:

- Reasoning accurately (having it so that intermediate steps in your reasoning process make a firm foundation for further steps - so as to decrease probability of doing steps that aren't useful or result in mistakes)
- Being good at concepts/abstractions (coming up with them, evaluating them, using them). Identifying various parts of your domain according to things that are true about them (similarities, structural overlap, relations and interactions with other parts of domain, etc), and finding patterns or statistical phenomena that (1) apply to these classifications and (2) are helpful to the search.
- Making effective choices for where/how to "look" when looking for the next "step" in the search-process. There is a huge possibility-space of stuff that can be done, which means that the system needs to have good procedures for (1) determining "where" and "how" to search, and for (2) continuously evaluating if a certain "search-path" seems worthy of continued exploration.
- Being good at evaluating whether a given result from the "search-process" fulfills the criteria that the search attempts to optimize for (not just making such evaluations at the end of the "search", but also throughout the "search-process").

Me: So narrowly intelligent AI-systems may often have big structural similarities. That is to say, they may be designed such that there are many sub-modules that either are exactly the same or have great structural similarities.

Me: Imagine for example an STV with the purpose of making code more understandable for humans. Well, in many ways that is a search process. The heuristics and so on for deciding where to look next in “possibility-space” may be domain-specific, but maybe the STV could be designed to start out without these domain-specific heuristics (and have domain-specific heuristics be generated based on processes that are less domain-specific).

Me: Maybe for some domains it’s hard for us to evaluate if the “search procedure” really is finding the best results that can be found. In such cases it may be an advantage if we can test that same “search procedure” (or “search procedures” that are as similar as possible) in other domains where it’s easier to evaluate the results.

Imaginary friend: Let’s think of “evaluation” and “search” as separate (even though there presumably is lots of inter-play). To put it simplistically, we have “search” and we have a “scoring-function” (or more complex interactions with modules that evaluate “score”). And for the sake of argument, let’s assume that we can verify that in some sense the “search” is “optimal” (in terms of being able to find options in possibility-space that are scored high by the “scoring-function”). Even **if** that’s the case, that still leaves the challenge of making a “scoring-function” that reflects what you actually want?

Me: Sure, that’s a challenge. And how challenging that part is will vary from STV to STV. The “scoring” of some solution is itself a task that other STVs could be specialized to work on. In many cases we may get far even if they don’t do a perfect job to begin with.

Me: Consider for example a scoring-function that evaluates how readable some piece of code would be to a human. Even if this function is imperfect, it will probably still help quite a bit. And if we noticed that it was missing obvious improvements, then this could be fixed.

Me: And as we gain more capabilities, these capabilities may be used to refine and fortify existing capabilities. For example, if we obtain STVs that are verified to do a good job of predicting humans, then these may be used to more comprehensively test and improve scoring-functions (since they are able to compare 2 different ways to write code with equivalent functionality, and can help predict which way makes it easier for humans to understand it and notice problems).

Thanks for now

Me: More things can be said about STVs and their potential uses, but I’ve talked for a long time now. Probably best to save other stuff for later.

Imaginary friend: I don’t disagree..

Me: The next part of this series will focus especially on computational proofs, and how we might use proofs much more extensively than today (blurring the line between computational proofs and rigorous arguments more generally, and dealing with vagueness and model-to-reality mappings).

Me: The part after that will talk more about chicken and egg problems, about strategic and moral considerations, and about possible first steps when using AGIs outside of the digital realm.

Imaginary friend: Ok, talk to you later.

…

To me the concepts/ideas in this series seem under-discussed. But I could be wrong about that, either because (1) the ideas have less merit than I think or (2) because they already are discussed/understood among alignment researchers to a greater degree than I realize. I welcome more or less any feedback, and appreciate any help in becoming less wrong.