Research as a Stochastic Decision Process

In this post I will talk about an approach to research (and other projects that involve high uncertainty) that has substantially improved my productivity. Before implementing this approach, I made little research progress for over a year; afterwards, I completed one project every four months on average. Other changes also contributed, but I expect the ideas here to at least double your productivity if you aren't already employing a similar process.

Below I analyze how to approach a project that has many somewhat independent sources of uncertainty (we can often think of these as multiple "steps" or "parts" that each has some probability of success). Is it best to do these steps from easiest to hardest? From hardest to easiest? From quickest to slowest? We will eventually see that a good principle is to "reduce uncertainty at the fastest possible rate". After revealing issues with more simplistic approaches, I will articulate this principle in detail and show how to apply it. Throughout, I draw my examples primarily from problems in machine learning and mathematics, but I believe that the principles generalize to other situations as well.

Warm-Up

Suppose you are embarking on a project with several parts, all of which must succeed for the project to succeed. For instance, a proof strategy might rely on proving several intermediate results, or an applied project might require achieving high enough speed and accuracy on several components. What is a good strategy for approaching such a project? For me, the most intuitively appealing strategy is something like the following:

> **(Naive Strategy)**
> Complete the components in increasing order of difficulty, from easiest to hardest.

This is psychologically tempting: you do what you know how to do first, which can provide a good warm-up to the harder parts of the project. This used to be my default strategy, but often the following happened: I would do all the easy parts, then get to the hard part and encounter a fundamental obstacle that required scrapping the entire plan and coming up with a new one. For instance, I might spend a while wrestling with a certain algorithm to make sure it had the statistical consistency properties I wanted, but then realize that the algorithm was not flexible enough to handle realistic use cases.

The work on the easy parts was mostly wasted--it wasn't that I could replace the hard part with a different hard part; rather, I needed to re-think the entire structure, which included throwing away the "progress" from solving the easy parts.

What might be a better strategy than the naive strategy above? Since the naive strategy has the problem that we waste effort on the easy components if the hard components are intractable,

maybe it would be better to complete the components in *decreasing* order of difficulty, starting from the hardest and moving to the easiest.

This *might* be better, but our intuitive sense of hardness likely combines many factors--the likelihood that the task fails, the time it takes to complete, and perhaps others as well. Here is an example:

> Task A is a detailed and tricky calculation, but you have done many similar calculations before and are confident that given a few days you will succeed. Task B will likely take much less time, but it is something you haven't done before (so it is more likely there will be an unforeseen difficulty or problem).

In this case, task B would be better to do first--if you do task A first and then B turns out doomed, you have wasted several days. Even if A also has some chance of failing (so that it is both more likely to fail and takes longer than B), we would still usually rather do B before A.

This reveals that harder tasks should not necessarily be prioritized. Rather, we should prioritize tasks that *are more likely to fail* (so that we remove the risk of them failing) but also tasks that *take less time* (so that we've wasted less time if one of the tasks does fail, and also so that we get information about tasks more quickly).

A Better Strategy: Sorting by Information Rate

We can incorporate both of the above desiderata by sorting the tasks based on which are *most informative per unit time*.

> **(Better Strategy)**
> Do the components in order from most informative per unit time to least informative per unit time.

To implement this, we need a method for quantifying informativeness. I will present two methods below--one based on *expected time saved*, and one based on *failure rate*. Rather than define these rigorously upfront, I will work through several examples, which should make the general case evident.

Method 1: Expected Time Saved

If an earlier step fails, we save time by not having to attempt the later steps. We should therefore complete the steps in the order that maximizes the expected value of the time that we save. We assume for now that we can actually quantify the probability that each step succeeds, as well as the time it will take. Consider the following example:

> Example 1: All of the steps of a project have roughly equal chance of success (80%, say) but take varying amounts of time to complete.

In this example we would want to do the quickest task first and slowest last, since the later a task occurs, the more likely we will get to skip doing it. Sorting “easiest to hardest” is therefore correct here, but it is rare that all steps have equal success probability.

> Example 2: An easy task has a 90% success probability and takes 30 minutes, and a hard task has a 40% success probability and takes 4 hours.

Here we should do the easy task first: if it fails we save 240 minutes, so 0.1 * 240 = 24 minutes in expectation; conversely if the hard task is done first and fails, we save 30 minutes, for 0.6 * 30 = 18 minutes in expectation. But if the hard task takes 2 hours or the easy task has a 95% chance of success, we should do the hard task first.

Thus, in this method we formalized “most informative per unit time” by looking at how much time we save (in expectation) by not having to do the tasks that occur after the first failure. Our computations assumed that we only find out if a task succeeds or fails at the end, as opposed to in the middle; however, they can be modified to take such complications into account.

For more than two tasks, this calculation method quickly becomes intractable: for K tasks we have to consider all K! permutations to find the best one. The next method avoids this issue.

Method 2: Failure Rate

This next method models the occurrence of failures as a Poisson process: if a task takes 30 minutes and has a 15% chance of failure, then there is about a 0.5% chance that the failure will occur in each minute (actually, it is slightly more than that because of overlap among the failures; the actual value is the solution p to (1-p)^30 = 0.85). Note that this differs from our previous assumption that failures can only occur at the end. This alternate model will simplify our calculations.

Formally, assume that the probability that we realize the task fails in the next minute is independent of how long we have been doing the task. Then the occurrence of a failure is a Poisson arrival process and the time at which a failure occurs follows an exponential distribution with some rate parameter λ, where λ tells us how frequently failures occur per unit time. Using basic properties of Poisson processes (see Appendix A), we can compute λ as

$$\lambda = \log(1/p_{succ})/T,$$

where p_{succ} is the success probability of the task.

This rate exactly tells us how quickly we will encounter failures while doing a given task. Since we would like to front-load failures as much as possible, we would always like to sort the tasks in decreasing order of their rate λ.

Returning to Example 2, we can compute the rate λ for the two tasks:

Task 1: $\log(1/0.9)/0.5 = 0.21$
Task 2: $\log(1/0.4)/4 = 0.23$

This new computation reverses our previous conclusion: The hard task has a higher rate, so is actually (slightly) better to do first! The reason for this is that the Poisson assumption implies that the higher the failure probability of a task, the faster (in expectation) we will encounter the failure. This contrasts with the previous assumption that we only encounter failures at the end of a task. We should keep in mind that both of these assumptions are likely somewhat incorrect in practice.

The rate method extends easily to more than two tasks, since we can simply sort tasks in order of $\log(1/p_{succ})/T$.

An Additional Example
In the case of the time-consuming but certain task A and quicker but uncertain task B, task A might take 12 hours but have a 90% chance of success, while task B takes 2 hours but has a 65% chance of success.

First, let's see what we get using the time saved method:

- A first: 0.1 * 2 = 0.2 hours
- B first: 0.35 * 12 = 4.2 hours

Now suppose we use the rate method:

- A first: log(1/0.9)/12 = 0.009
- B first: log(1/0.65)/2 = 0.215

B dominates A on *both* failure prob and time, so doing B first looks substantially better under both methods.

Caveats
These numbers are all completely made up and in practice you won't be able to estimate things so well. I subjectively distinguish between different "buckets" of success probability, such as:

- "I am confident that this can be done and that there are no unforeseen difficulties" (~95%)
- "I am confident that this can be done modulo Murphy's law" (~90%)
- "I see the basic path to accomplishing this and all the steps seem like they should work" (~65%)
- "I have the intuition that this should be possible but only have a murky view of the path" (~30%)

On the other hand, I tend to have much better estimates of task completion times if I've been practicing (~30% average relative error, albeit with large tails). You can get better at this within a few weeks by estimating completion times for each task and then recording the actual completion times in a daily log. You should also practice decomposing tasks into small actionable chunks, each taking roughly 20 minutes to 2 hours.

A further improvement: opening up the "task" black box

Sorting tasks in decreasing order of failure rate is a good start; it should improve efficiency by a factor of 2-3. However, we can do *much* better still by learning to front-load the information gained about each task. Front-loading information requires a mental finesse: rather than seeking to complete a task, we must seek information *about* a task.

Specifically, for each task we want a *cheap way to obtain high confidence about whether that task will be feasible*. This is called "de-risking". The following pattern is indispensable:

> **(Basic Pattern)**
> De-risk all components (to the extent feasible), then execute.

As an example, suppose we wish to set up a dataset and then train a suitable model on that dataset. However, setting up the dataset is arduous: we must download it to a place with enough disc space, parse it into a usable format, and incorporate auxiliary data sources (like noun/verb banks for natural language processing).

Setting up the dataset and training the model are both time-consuming and either one could fail. Even worse, it would seem that we are forced to set up the dataset first, even though it is probably the more time-consuming task.

To avoid this issue, we could first download a few thousand examples. We can then examine several examples by hand, as well as compute some aggregate statistics, to assess whether the dataset has the properties we want. Ideally, this will reduce a lot of uncertainty about whether the full dataset is suitable.[1]

General principle: stochastic decision process

We can unify and extend the above insights by modeling a research project as a *stochastic decision process*. Specifically, we think of research as a multi-round game, where in each round we take some action that gives us some information; the information we get is stochastic, and well as perhaps the time needed to complete the action. We have two competing goals:

---

[1] This doesn't quite fit into the framework because if the dataset is unsuitable we can try again until we find a suitable dataset. But it could be that we try 4 datasets, they are all unsuitable, and we eventually conclude that there aren't any suitable datasets. The sort of de-risking above allows us to reach this conclusion much faster and avoid spending time trying to train a model on a broken dataset.

- Maximize probability of eventual success (don't give up if it turns out we can eventually solve the problem).
- Minimize expected time spent (give up early if the problem is not feasible, and solve the problem quickly if it is feasible).

We are often either in "de-risking mode" (determining if the problem is infeasible as quickly as possible) or "execution mode" (assuming the problem is feasible and trying to solve it quickly).

An aside: tooling. This picture grows more complicated if we consider actions that could speed up a family of future actions (such as writing helpful scripts to automate tasks, or reducing the execution time of the system). Such "tooling" tasks are tricky to model, because it seems we should implement tooling as soon as we know we will eventually want it (since it speeds up things that come after it). However, this ignores that more experience often yields refined desiderata for the tools we implement. There is thus a trade-off between building tools earlier vs. building better-targeted tools. I won't say more about this here, but it is an important point to keep in mind.

Another complication is that our ultimate goal is often nebulous--we are not asking "is this problem possible" so much as "how interesting of a problem in this space is it feasible to solve"? But I don't think this substantially alters the above principles.

Some further practical ideas

There are a number of useful patterns for putting the above principles into practice. I list several below.

For empirical work, measuring "ceilings" (an upper bound of how high performance could possibly be) is often useful. Example: suppose we wish to build a system with 3 components that interact in a complicated way. One of the components is difficult to implement, but we can easily substitute a "cheating" version of that component (e.g. by looking at the test set or by using information that won't be available at deployment time). We often benefit by building a prototype system that initially uses this cheating version:

- If the system works, we know that a sufficiently good implementation of the difficult component will yield a working system.
- If the system doesn't work, we've saved the time of implementing the difficult component.

We can choose which components to cheat on initially, and which to implement fully, using the "informativeness per unit time" heuristic from above. For instance, if the ability to do well on a specific component is the major source of uncertainty in the project, cheating on it might be counterproductive (we may instead want to cheat on *everything but that component*).

The counterpart to ceilings are *baselines*--simple or off-the-shelf methods that give a quick lower bound on achievable accuracy. Baselines provide an important sanity check, as complicated methods often underperform simple baselines. Together with ceilings, they delineate a range of possible performance, which helps us interpret our core results.

Brute force. If we know of an easy-to-implement brute force solution and a difficult-to-implement fast solution, starting with the brute force solution has many of the same advantages as using ceilings, as long as the slower running time doesn't bottleneck prototyping. A brute force implementation also facilitates debugging the fast solution, since we can compare the outputs of the two algorithms.

As with ceilings, brute force is most useful when implementing the fast solution is not a major source of uncertainty (e.g. it is routine but annoying, or is one of many sources of uncertainty).

For theoretical work, looking for counterexamples is useful. The simplest example of this: if we find a counterexample to the main result we want to prove, then we need to either give up or make stronger assumptions.

A more nuanced (and more common) example: if we are trying to prove that $A \implies B$, and our current technique does this by proving $A \implies X$ and then $X \implies B$, finding a counterexample to $A \implies X$ will rule out that technique.

Yet more nuanced/common: if we are trying to prove that $A \implies B$, and our current technique applies equally well under assumptions A and A', then a counterexample to $A' \implies B$ will rule out the technique.

More generally, thinking about simplified instances of a problem is often useful. This is because it provides intuition that often suggests/rules out approaches for the original problem. Similarly to de-risking, the ability to rule out entire approaches makes this tactic invaluable from the stochastic decision process perspective.

Running simulations. If we wish to prove X, first run simulations to check if X is actually true. This is easy when assessing the behavior of a specific algorithm, as we can simply run the algorithm. Simulations can also, for instance, help reveal the asymptotics of a random process, or be used to search for small counterexamples to a conjecture.

Exponentially branching search trees

Another important mental framework focuses on the combinatorial aspect of a decision process:

> **(Research as branching search)**
> I often think about possible approaches to a problem as an exponentially branching search tree: we could try X, X', or X''. Then X could be combined with Y, Y', Y'', or X'

could be combined with Z or Z', etc. This exponential blow-up poses barriers to projects with more than a small number of steps unless we have a way to systematically rule out entire branches of the tree.

Exponential branching often occurs because there are many ways to try a particular approach--perhaps we want to bound the moment generating function, and there are many ways to attempt this; or we think data augmentation will help our model generalize, but there are many ways to augment the data. With many possibilities for each step, even a two- or three-step approach creates a huge search space. For instance, if there are 10 ways to try bounding the moment generating function, and two other similar steps, then we have to try 1000 possibilities.[2]

If the steps *factor*--meaning they can each be solved in isolation--this might be fine (we only have to try 3*10 instead of 10^3 possibilities). However, I usually find that there is some interdependency between different steps. For a math problem, maybe how good of a bound I get from step 1 affects how hard I need to work for step 2. Or for an experiment, if any of 3 parts of the setup are wrong then the method just won't work, so I don't get signal until I've gotten a few things right simultaneously.

For this reason, I think it's *much* more useful to prune branches of the search tree at the level of conceptual approaches ("can the moment generating function give me sufficient control over the distribution I care about?") than at the level of a specific instantiation ("does this particular moment generating function bound work?"). This leads to adopting several principles:

Whenever something doesn't work, I ask *why* it didn't work. My goal is to avoid trying similar things that will fail for the same reason (or to notice that the reason why it didn't work is circumventable, and that a modified approach actually will work).

Trying an experiment and seeing it fail gives little information by itself. When an experiment fails, it is tempting to conclude "I tried X and it didn't work". However, if X is a high-level conceptual approach, then a more correct conclusion is "I tried an implementation comprising 0.1% of the possible implementations of X, and observed that that particular implementation did not work". For this reason, I am far less in favor than most people of publishing negative results, unless the negative result comes with insight into what caused the failure. In contrast to common concerns, negative results that come with such insights are already publishable.

Compared to other people I know, I try harder and earlier to show that my ideas can't work to solve a problem. Importantly, it is often not obvious that multiple approaches to a problem all have the same issue. In the past, I have spent months trying different approaches to

[2] This is purely illustrative and in reality we can't necessarily decompose different attempts into a fixed number of discrete "ways" of attempting something.

a problem before finally stepping back and realizing that they were all failing for the same reason. Moreover, I had all the data necessary to make this realization a couple weeks in but had failed to do so. I now save considerable time by ruling out ideas early on, and as a result I am usually bottlenecked on coming up with ideas rather than on implementing ideas.

Additional Discussion

In the previous section I talked about ruling out ideas. When ruling out ideas, it is important to hold oneself to a high standard. “This doesn’t seem like it will work” or “I feel less motivated after trying a few things along this line that didn’t work” are *not* ruling out an idea. We could perhaps think of them as updating the probabilities that a solution lies within a given subtree of the search tree. But these updates are rarely large updates, and I find them much less reliable than a solid argument for why an approach is doomed.

Note that I am *not* advocating that you should never trust your feelings. If you feel pessimistic about an approach, that is a great reason to try to show that the approach can’t work! If I feel pessimistic about an approach but fail to rule out that it could work, I often then feel more optimistic.

I am also *not* advocating for the failure mode of only trying low-variance ideas, or of avoiding projects that lack an obviously promising approach. Part of the point of being able to systematically rule out ideas is to enable trying ideas that only have a low probability of working, or that do not immediately yield progress.

Summary

Many of our default intuitions about how to pursue uncertain ideas are counterproductive:

- We often try easier tasks first, when instead we should try the most informative tasks first.
- We often conflate a high-level approach with a low-level instantiation of the approach.
- We are often too slow to try to disprove our own ideas.

Building frameworks that reify the research process as a concrete search problem can help unearth these incorrect intuitions and replace them with systematic reasoning.

Appendix A: Poisson Process Calculation

In a Poisson process with rate λ, the probability that a failure has already occurred by time t is $1 - \exp(-\lambda t)$, so in particular $\exp(-\lambda T) = p_{succ}$, where T is the time to complete the task and p_{succ} is the success probability of the task. If we solve for this, we get that the rate λ is equal to

$$\lambda = \log(1/p_{succ})/T,$$

as claimed.