Spring 2016

In the Face of Anticipation: Decision Making under Visible Uncertainty as Present in the Safest-with-Sight Problem

Bryan A. Knowles
Western Kentucky University, bryan.knowles951@topper.wku.edu

IN THE FACE OF ANTICIPATION: DECISION MAKING UNDER VISIBLE UNCERTAINTY AS PRESENT IN THE SAFEST-WITH-SIGHT PROBLEM

A Thesis
Presented to
The Faculty of the Computer Science Department
Western Kentucky University
Bowling Green, Kentucky

In Partial Fullfillment
Of the Requirements for the Degree
Master of Science

By
Bryan A. Knowles

May 2016

IN THE FACE OF ANTICIPATION: DECISION MAKING UNDER VISIBLE UNCERTAINTY AS PRESENT IN THE SAFEST-WITH-SIGHT PROBLEM

Date Recommended 3-15-2016

[illegible]
Mustafa Atici, Director of Thesis

[illegible]
James Gary

[illegible]
Qi Li

[illegible] 4/18/16
Dean, Graduate Research and Studies Date

Contents

IN THE FACE OF ANTICIPATION:
DECISION MAKING UNDER VISIBLE UNCERTAINTY
AS PRESENT IN
THE SAFEST-WITH-SIGHT PROBLEM

Bryan A. Knowles May 2016 118 Pages

Directed by: Mustafa Atici, James Gary, Qi Li

Department of Computer Science Western Kentucky University

Pathfinding, as a process of selecting a fixed route, has long been studied in Computer Science and Mathematics. Decision making, as a similar, but intrinsically different, process of determining a control policy, is much less studied. Here, I propose a problem that appears to be of the first class, which would suggest that it is easily solvable with a modern machine, but that would be too easy, it turns out. By allowing a pathfinding to anticipate and respond to information, without setting restrictions on the "structure" of this anticipation, selecting the "best step" appears to be an intractable problem.

After introducing the necessary foundations and stepping through the strangeness of "safest-with-sight," I attempt to develop an method of approximating the success rate associated with each potential decision; the results suggest something fundamental about decision making itself, that information that is collected at a moment that it is not immediately "consumable", i.e. non-incident, is not as necessary to anticipate than the contrary, i.e. incident information.

This is significant because (i) it speaks about when the information should be *anticipated*, a moment in decision-making long before the information is actually collected, and (ii) whenever the model is restricted to only incident anticipation the problem again becomes tractable. When we only anticipate what is most important, solutions become easy to compute, but attempting to anticipate any more than that and solutions may become impossible to find on any realistic machine.

Chapter 1

Introduction

The Unknown is to be discovered. The Unknown excites us. Imagine–the stands are full, thousands of fans, their foam fingers in the air as they cheer their team on. No one, save for Lady Luck, *knows* who will win the game. And when the final score is displayed high up on the board, a series of lit and unlit lightbulbs, some broken, but the message still clear, the winning team will attribute their victory to skill, hard work, and dedication. The losers will curse their luck. They will blame the angle of the Sun or maybe haze the junior players for not following to the key the pre-game rituals.

In the audience, the statisticians watching view the game with *rigor*. They model the outcome as a result of a *random variable*, a number selected from a basket following the well-defined rules of a *probability distribution*, a mathematical function that assigns a likelihood to each possible outcome. Likewise, the dieticians and biologists ponder the meals the team may have had for breakfasts, the dinners, and exercise regiments consumed in preparation for the big day. They hypothesize about the effects of carb and protein intake on the players' energy reserves and mental alertness.

The Unknown tempts us in this area of game-playing because we know, by some specific time in the future, that we will come to learn the outcome, to see what The

Unknown has in store for us. The contest can only last for so long–even in the event of multiple overtimes–, after which the victor will be certain. It will then be clear. It will then be known.

But The Unknown can also exhaust us. What if the "game" is one that may never end, when the answer is one that may never come? Think of the astronomers who dedicate their lives to counting the innumerable stars. No one knows *when* another planet will be discovered, or even who will do the discovering or how they will do it. Perhaps it will be through the development of a new class of space telescope. Perhaps it will be deduced that a new planet exists past our visible horizon by how its gravitational pull is affecting the smaller heavenly bodies that we can see.

Despite these uncertainties, the world of science dedicates its life to braving such frontiers. Perhaps it is this relentless drive for discovery that lends itself to the storytelling archetypes of the *eureka!* moment and the *mad scientist*. Scientists have bad hair: there is no time to waste on such trivial matters and they are always running their fingers along their scalp as they ponder depths of The Unknown. Scientists drink coffee from beakers: they have so surrounded themselves in their studies that the line is blurred between home life and work. Scientists are benevolent: they seek only to change the lives of their fellow humans for the better. Separately, we have *evil scientists*, in which case they instead seek the power to bend the natural world to their will.

These are, of course, overgeneralizations, Hollywood movie stereotypes. If the tropes are accurate, then I've never even met a scientist. Who've I've met instead are curious people, people comfortable, or at least complacent with, the academic funding model. I've met people who want to solve and to teach.

It is only when we seek absolutes about the Universe, certainties about our fates, or trivialties about intractable problems, incessantly, that we fall into the scholarly madness that popular culture holds to be true. I have untaken such a task. The difficulty was accidental, much like how carving through of Hoosac Mountain was illusory [8]. Geologists predicted that the multi-million dollar project would be complete much sooner, much more easily, and for much less cost than The Unknown would have it be. I was intruigued by a subject far less grand than mountain carving though. I just had a dumb question about sidewalks.

1.1 A Clear Road Lies Ahead

In the summer of 2012 I was mid-way through my courses at Western Kentucky University for a Bachelors in Computer Science. Having completed the manditory two years' dormatory residency, I moved into a house, accompanied by four roommates I had known since high school. The house was close to campus, only 1.2 miles south from my old dorm. I borrowed a bike from my then-sweetheart and started my days crossing through campus on my way to my job as a web developer. My legs soon felt what old men talk about when they use the cliche, "uphill both ways."

Summer, that summer in particular, was the season of reconstruction at the university. The central building, Downing University Center, was being renovated, piecewise and in whole. It was to be renamed two years later as Downing Student Union. Repair work to the schools infrastructure–power, water, waste–was also slated to be carried out at the time. But any reconfiguration, in any system, must be done at the temporary cost of some resource. For example, cleaning the house may require waste to be accumulated in one area, consuming that once free space, until it can be removed from the home. Like sweeping. Being the only clean one of five roommates soon makes apparent any restrictions of already clean space to further clean with.

Sidewalks were the obvious such temporary resource at WKU. With pedestrian traffic reduced to negligible numbers, utility workers were free to lift out sidewalks to access the infrastructure lines lying beneath, to park their work vehicles on the sidewalks to keep the roadways clear, and to remove sidewalks altogether, curving the pedestrian traffic differently to better suit the buildings being added or torn down.

Change was happening at Western Kentucky University, and I was biking through it twice a day, uphill both ways, weekdays. Keeping to the sidewalks kept the tires spinning with the least physical exersion, but the sidewalks that were "up" and "down" changed each day.

The maintainence crews moved quickly. I knew the particulars of WKU's pathways by heart, and yet I failed, several days in a row, to find a path from one side to the other, a path that did not require me to go offroad, in the road, or turn my vehicle around.

I asked myself, "What is the path that *maximizes* my chances of cutting through campus, on bike, without leaving the sidewalks or backtracking?" The climb up–the carving through–my mountain of difficulty began, innocently, then. This problem plagued me incessantly, so it follows that in a few years I too would certainly be a *mad scientist.*

1.2 Bridges, and How to Cross Them

It turns out that finding the safest route through campus is pretty damn trivial.

Imagine, for example, that I've spread throughout campus a bucket's worth of special coins. Each time you come across such a coin, I challenge you to a coin tossing game. If you flip heads, you may pass. But if you flip tails, I break your knees, like in a classic mafia film with a baseball bat, duct tape, and two chairs. It only makes sense that you should take the path with the fewest coins; I'll even let you know in advance exactly where each coin is, just to make things easier for you. And your knees.

Graph theory is already well-equipped to handle this problem. This is the study of points and the lines that connect them. The origins of this subject area is usually attributed to the Konigsberg Bridge problem, which asks if there is a way to cross each of the city's bridges–those of Konigsberg, a German city–exactly once, without having to swim through the river or take to flight, to put it in networking terms. The regions of the city could be taken as points and the bridges as their inconnecting lines. Next, consider the *degree* of each point (region). This is the number of lines coming into or out of each point. If there is one, three, or more points (but not two) with an odd degree, then a path that crosses each bridge exactly once is impossible.

The proof is easy. To enter a region of the city, we must cross a bridge, removing it from our list of bridges we can, and must, still cross. To then leave that region, we again cross a bridge and cross it off our list. In this way, we can remove exactly two bridges from the list. Passing through the region again marks off another two bridges. And again, another two. If we pass through the region k times, $2k$ bridges are eliminated.

So what if there is a region with $2k+1$ bridges connecting to it? If we pass through it k times, we'll still have exactly one bridge left. If we try to pass through it again, we'll become *stuck* in that region. There will be no bridges left for us to leave by.

But there is a special case: if there are *exactly two* points with odd degree, we might be able to leave one and enter the other, resolving the odd-difficulty, but one, three, four, or seventeen points with odd degree and we've become trapped again.

It is this sort of problem that graph theory is most comfortable solving. In the formal parlance of the theory, we say a graph (city) G is composed of two components, a set of vertices (regions) V and a set of edges (bridges) E, or more commonly put, $G = (V, E)$.

If we add extra information to our graph, like distance, the applications quickly expand, and a famous algorithm for solving the "shortest path" problem, Dijkstra's Algorithm, is exactly what we are looking for to solve the "special coin" problem.

1.3 A Man Named Dijkstra

First, determine the "distance" of each edge (sidewalk). It's simplest to just let that be the number of special coins spead out along that length of sidewalk. Then, apply Dijkstra's Algorithm, which seeks to find the shortest path from all points to one point in particular, the "starting point," thus minimizing the number of special coins you must flip.

```julia
function Dijkstras(G)
    unvisited_vertices = Int[]
    distances_to_start = Int[]
    previous_vertex = Int[]
    for i in vertices(G)
        push!(unvisited_vertices, i)
        push!(distances_to_start, Inf)
        push!(previous_vertex, Inf)
    end
    distances_to_start[1] = 0
    while length(unvisited_vertices) > 0
        i = remove_closest!(unvisited_vertices, distances_to_start)
        for j in neighbors(G, i)
            alternative_distance =
                distances_to_start[i] +
                direct_distance_between(G, i, j)
            if alternative_distance < distances_to_start[j]
                distances_to_start[j] = alternative_distance
                previous_vertex[j] = i
            end
        end
    end
end
```

The above code is in the Julia programming language. This language is chosen because of its balance between expressiveness–for my sake–and its readability–for your sake. This code shows an example of how we can write out Dijkstra's Algorithm. In English, we start with a very bad guess of the overall distances between points and update it, one step at a time:

1. first we say that the total distance from all points to the starting point is

infinity–no number could be worse;

2. next we say that the distance from the starting point to itself is zero–an obvious statement really;
3. we then "visit" each vertex, one at a time, each time selecting to visit the vertex closest to the starting point, according to the total distances we have written down so far;
4. the first vertex to be visited will be the starting point itself, since everything else is as far away as can be–zero is much, much smaller than infinity;
5. then, for each neighbor of the visited vertex (those connected directly to it) we ask ourselves the question, "is it better to go from the starting point, *through* the visited vertex, then continue on to the neighboring vertex, or is it better to just keep whatever path we've figured out so far for that neighboring vertex?";
6. if our answer is yes, then we update the information we have stored about what paths are best to follow;
7. and so on, in this incrementally-improving fashion, we go from distances of infinite length to paths of shortest distance.

The time it takes for Dijkstra's Algorithm to find all the shortest paths to the starting point is, computationally speaking, small. The number of machine operations it takes for the `for` section to complete is proportional to the number of the graph's vertices, and the same goes for the `while` section. So, if we have n points to find paths through, it will take a computer something like $2n$ steps to find the answer.

Consider, if n is of any concernable magnitude and it takes $n!$ or 2^n steps to solve the problem, regardless of how fast of a computer the NSA has, the Sun will explode long before it will respond. The human race and the whole of our solar system engulfed in an unimaginable heat-bomb, and it will be the government's fault–because they chose a subpar algorithm. Grimly put, this notion is still at the heart of computer science. It leads us to search for stars of our own, the efficient algorithms, the ones that require no more than n^k steps: the *polynomial* algorithms.

If I leave this safest route problem as is–a search for a fixed path that minimizes the number of opportunities I have to break your knees–, then it is said to have a solution in polynomial time and our work is done.

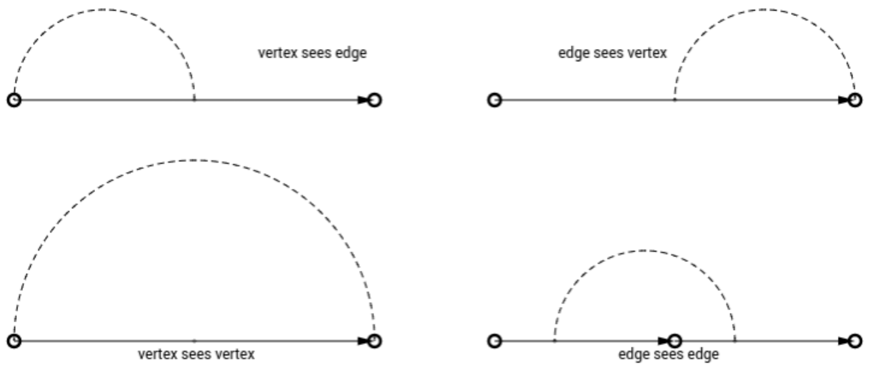
But I'll do no such thing. It turns out I can create a problem whose solution requires that forbidden 2^n steps by adding two new rules:

- you can use your eyes to see dangers ahead of you; and
- you can "reroute" yourself, mid-course, around these.

1.4 A Free Tour of the Multiverse

Graph theory is no longer equipped to solve this new problem on its own.

The addition of vision complicates things because not only are there just vertices and the edges that connect them, but also there's "lines of sight" that connect points to the edges. We can "see" edges from vertices through these lines-of-sight. Equivalently, as far as computation is concerned, we could say that vertices provide vision of other vertices, that edges provide vision of other edges, or that edges provide vision of vertices; these formulations are strange, hard to imagine, or needlessly complicated. We could just transform any of these alternatives back into the herein-preferred "vertex vision of edges" definition anyway and keep the language human-readable.



These transformations–in computer science speak, *reductions*–become apparent when we spend some time viewing the world from different angles. First, imagine the grid-like roadways of Manhattan as a graph, as points (intersections) connected by edges (roads). At each intersection, our driver brings her taxi to a stop, turning her head from left to right, back to the left, and back to the right, gathering information about the nearby traffic situation. Perhaps her phone will ring and, in a short conversational burst, she will learn from her caller about a wreck further down towards our destination. In this view, we only receive information about the city while stopped at an intersection (and we'll assume that Manhattan is as Manhattan is and we hit every possible red light). Also, we only learn about the traffic along the roads, not at the intersections. This Manhattan is just like the "vertex vision of edges" version of the safety problem.

To reduce this to an "edge vision of vertices" formulation, consider how we chose to represent the city as a graph: instead of letting vertices represent intersections and edges represent roads, as is natural, we can let vertices represent roads and edges represent intersections. To see this new world, imagine a map of Manhattan. Picture yourself circling a point along each road. This point (vertex) will represent that road in its entirety. Then, every time that you see that two roads share an intersection, draw, right on top of the map, a line (edge) connecting the points that represent those roads.

With some stretching of the imagination, we should be able to see that this "inside-out" view of Manhattan is possible, even if it is contrived. Computer science, when speaking about reductions, is not concerned with whether our definitions resemble the real world anymore, but that it's possible, in any manner, to translate from the language of one problem to another and back again. Here, we have just shown that possibility, so we continue.

To reduce to the "vertex vision of vertices" case, we first examine how two vertices are already connected, by a shared edge, in "vertex vision of edges." Instead of drawing two points connected by a single line, we draw the same two points connected by the same line, but then *break that line in two.* These two new edge-pieces will be connected by a new point added to the mix. In this version of the world, there is no traffic. Instead, there is a police officer standing in the middle of each road, directly between each pair of neighboring intersections. This police officer is either mad–he hadn't had his coffee that morning–, in which case he will not let us pass. Otherwise, he is happy and lets us on by with a wave and a smile. And, instead of looking left, then right, then left again to see how traffic is flowing, our driver now looks around at the police nearby, reading their facial expressions when able, taking us along a route filled only with smiles or, when that isn't possible, along a route filled only with smiles and officers whose backs are turned to us. Since we can't read them clearly, passing these officers would regretfully be a gamble.

This police state version of the world is also reducible to "vertex vision of edges"– all the worries we had regarding the edges, those that were once about traffic or torn out sidewalks, have been transformed into worries of equal magnitude, those about which of the city's "boys in blue" had a worthwhile breakfast.

The final reduction is now much simpler than the others to state, it so happens. Reduce the preferred "vertex vision of edges" version of the world into that of the police state version just discussed; then, just as we flipped the world "inside-out" to derive the "edge vision of vertices" formulation, we flip the police state world inside-out, leading us directly to a strange, strange "edges vision of vertices" world. An inside-out, police state world.

As such is computer science. Frequently, it is more efficient to solve a problem by transforming it to another problem for which an efficient solution is already known. Apply that solution, then transform the result so it again applies to the original problem of interest. Along the way, though, we may find ourselves operating in "uncomfortable dimensions."

Herein, we have chosen to work strictly with a definition of our safe-passage problem where, upon reaching each intersection, the cab driver is given information about the city's roads. This choice is made without loss of generality; we could easily apply this same language for solving problems where, for example, a "vertex vision of vertices" representation would be more natural. Consider a computer network. The machines frequently *ping* one another, keeping tabs on whether their neighbors–those they have a direct connection to–are still online. A message, just as the taxi driver in our previous examples, travels from node to node along those connections, but instead of considering whether the connections are still intact, it checks whether the nearby nodes along its current trajectory are operational.

We hope that the applications of a solution to our sidewalk problem are becoming more apparent for the reader. The original inspiration was, admittedly, silly in nature; however, translating real problems into the "language of sidewalks" can lead to an improved safety in several domains. To name a few, consider networking, GPS navigation, operations research, economics, and autonomous vehicles–which Apple, Google, and other large companies have been "getting behind" for years now [13].

Fittingly, then, although graph theory will no longer suffice, now with the addition of vision into the problem, we can reach into "mathematically nearby" domains for assistance, a task we attend to in the following chapter.

1.5 Planning Ahead

The second addition to the "special coin" version of the problem, that you can change your mind as you go, does not complicate matters in the same manner that the previous addition did; no, this changes the class of problem we are facing altogether.

We aren't searching for a safest-path through campus anymore. The course that you plot might change frequently–hear, if you've ever heard one, a GPS navigator saying that dreaded phrase, “Recalculating,” repeating it at every single intersection as it is provided incremental knowledge about the city.

We now need a safest-*policy*, as Niknami put it in his 2014 preprint [15]. We need a way to determine, quickly if we can, at each step what *one next step* should be taken in response to the new information that is expected to be gathered. This policy captures not just one path–it captures all paths that might be traveled and when they should be traveled. It captures all of reality itself. It explores all of The Unknown, searching ahead for us, laying out a clear sequence of events in the language of “if this, then do that.”

But calculating such a structure is no trivial task. It requires first tackling a notoriously difficult idea, from statistics to physics, operations research, and computer science–*expected maximum*. This is, to put it in terms of a Dungeons and Dragons example, the average value we can expect to receive if we are taking the best of rolling several d20s. Obviously, if we roll a single die, our expected maximum will be just the mean value of the integers $1, 2, \ldots, 20$, which is 10.5. As we increase the number of dice, we increase our opportunities for improving this value.

Sidewalks are not dice. Before I decide to head north down one sidewalk instead of heading east down another, I must first consider how I would behave once I reach my next step in either case. I know that at each step I will choose the “best” option, the one that maximizes my probability of successfully reaching my destination. However, I don’t know what those probabilities are. Once I take either step, I will learn more information about the status of the campus sidewalks, more about which have been parked on or torn out or otherwise obstructed. So, I don’t yet know what my future actions will be–it all depends on what I learn.

Instead consider the *expected maximum* success of my next step. At each intersection, I determine which step I can take next has on average the best follow-up step. I then take that step, collect whatever new information fate has in store for me, and lather, rinse, repeat. The difficulties of actually doing this, however, of actually coming up with expected maximums, are icebergs.

Without knowing, right now, what information values I will learn after my next step, even if I know exactly which edges I will be provided vision of, I am forced to resort to considering, right now, all possible outcomes and what my response will be to each. If I will be learning about n edges next step, I have to think about 2^n possibilities–2 for the first edge, which might be “up” or “down,” times 2 for the second edge, which also might be up or down, times 2 for the third edge, and so on, leading to n's exponentiation.

But even if I try to make the problem easier by saying I will only ever learn about, say, some small number h of edges at each step, so that 2^h is of an acceptable magnitude, I encounter another problem when I start to think about *the number of possible routes* I might take.

In the event that each path I can take to each intersection provides me with a different set of information–a worst possible scenario, but in the art of designing algorithms for solving problems, these sort of things must still be considered–, I am forced to work out the “what ifs” of each possible path. In a strange, strange world where every single intersection connects to every other intersection after it, there are 2^{v-2} possible paths, each beginning at the starting point and terminating at the destination, where v is the number of intersections. For a campus of limitless size, the significance of the -2 in the exponent falls away and we again must deal with computing, altogether, something akin to $2^h \times 2^n$ steps.

Our villian has reared his ugly head, and his name is $2 \times 2 \times \ldots$.

1.6 Horizons

Operations research, to my knowledge at least, does not typically concern itself with problems that are directly about pathfinding. It does, however, possess the language for developing mathematically-grounded policies.

Each day, week, month, quarter, or so on–in general, each *timestep*–, a manager is expected to make a decision, say, to determine how many of a particular item to order for the next timestep. Afterwards, fate takes its turn: weather and world events are moved into play and buyers come and go. Then, at the next timestep, the manager makes a similar decision, deciding how much of that item to have in stock for the next round, and the "game" continues.

It is common to purposely think of this as a game, a game played between the manager, who makes a move by setting the value of some variable, u, and Nature, a cruel mistress who is only out to hurt the company's profits, or Lady Luck, who plays totally unpredictably, in either case responding by setting the value of some other variable, x.

The goal of the manager in this game is to maximize the profits of the company, an "objective function," J, which observes the moves made by each player at each step and returns with a calculation of how much cold hard cash would be made.

Generally, there are two primary versions of this view of the world: the *infinite horizon* problem, where the two play their game on into forever; and the *finite horizon* problem, where upon some agreed date the game will end and the players will cash in their coins. Campus is, in a sense, much more like the latter problem–there is a limit to how long I can wonder around if I'm not allowed to visit the same place twice.

But there is a third, much less researched view, one that is significantly more readily related to our safest-policy problem: the *rolling horizon* [16] problem. In this, the decision maker is given, by some mechanism, advance knowledge of the next few moves that Lady Luck will make; the game may continue indefinitely, but the forecast information made available is always finite.

In the four years that I've so far spent on this problem, I've yet to discover a model of *any* problem closer to this one than rolling horizon. But even it is incomplete for what we need: the number of timesteps in rolling horizon is always either fixed or infinite, but the number of steps I will make to cross campus will certainly vary, contingent on the information I will gather as I go. I could "pad" the number of steps I take by saying that, upon reaching the destination, I will walk in circles for as many steps as necessary to ensure that all possible paths through campus are of equal length. This, however, is awkward, contrived, not lending itself to any useful interpretation of the problem.

What it does make easier, though, is imagining a larger, more encompassing, albeit more theoretical problem, one that both our safest-policy and the various horizon problems are reducible to. And, fittingly, the equation that governs it is pretty.

1.7 Dissolving Uncertainty

A common image in storytelling, from the Illiad to Return of the King, is that of the prophet. The hero does not know what he will learn when he reaches the oracle's door and gained her favor, but he knows what he will ask her about. "What weakens my enemies," he says, just as a child might ask the Magic 8-Ball, "Why doesn't mommy love daddy anymore?" As we, the readers, travel alongside the protagonist on his journey up the mountain to seek answers, we, just as he, start to anticipate what we will learn. It is that anticipation that keeps us on our toes, that keeps us reading the murder-mystery until the very last word, that keeps us *invested.* This anticipation, too, is the common thread conneting safest-path and finite horizon.

Before I begin my trek across campus or the manager makes her first managerial decision, I do not know the values that the future holds. But I am not lost. Mathematically, we can say that this information–everything there is to know about the future–is held within a variable X. This X is not one value, but a set of values, each random, chosen by Lady Luck according to some rules that only she truly knows and that we may approximate with our models, our probability distributions, though we will fail to capture *every* nuance of the problem.

If we were to ask the oracle at the onset of our journey what the future holds, she may tell us all there is to know about X, divulging that X, the perfectly unknown, is truly x, the perfectly known. In this case, we can live our lives according to the following equations:

$$L_i(x) = \max_j \left[\phi_{ij}(x) + L_j(x)\right]$$

$$L_d(x) = 0$$

Here, L will tell us how "good" our lives will be if we always make the best choices. At some moment in time or place on the campus graph i–the distinction between space and time does not matter here–, we choose, from all of options available to us, a choice j. We move towards j, whatever it represents. Perhaps j is a future life for the manager wherein she made some choice, chose some particular number of plastic forks to reorder for her restaurant chain; or perhaps j is chosen from the paths leading north, east, or south, those available to me during my campus-sidewalk-traversal. Whatever j represents, it is the choice that maximizes the "goodness" of the next step. A function ϕ_{ij} measures how good of a life will be lead if j was chosen from point i and the oracle had told us x.

And so on, for each step, we choose from all available next steps and collect whatever is due according to ϕ_{ij}. We only cease our decision making upon death, reaching the destination on the other side of campus, represented by L_d. We set this to zero: life is thus over, a life measured only by the decisions made during it. Mathematically speaking.

Several particulars are hidden away by the use of ϕ, which can only be determined by experimentation or reasoning, the goal of the researcher being to select a definition that makes this view of optimization one that suitably represents the problem they face.

But do not dwell on ϕ long. It is general and magical. Dwell, instead, on the oracle's willingness to provide us with a clear picture of all The Unknown, all at once, for no cost, up front. This will not do, for it does not resemble the real nature of things.

So imagine that at each step, the oracle provides us with new, but partial information. And, just as I know what sidewalks I will see as I round the corner and the manager knows what questions were asked on the company's latest consumer survey, we know what variables will be prophesized–although we know not, yet, their values. In a 1991 paper, Sethi posed a similar problem, rolling horizon [16]. That framework allows the manager to *purchase* information. Rolling horizon also found its way into Dimitriadis's work, in 1997, and Godfrey's, later in 2002 [6, 9]. These works were, respectively, concerned with factory scheduling and fleet deployments. Our problem is based on the notion that information is collected freely, yet strictly in accordance with what information could be naturaly gathered via lines-of-sight. If you can't see it, you can't see it.

Our variable X is this *free, but limited* information, and it was important earlier that X be a *collection* of values; this allows us to say $X_{(i)}$, the subset of that collection that we know will be divulged to us upon reaching i. With this particularity in hand, we can approach a modified expression of L.

$$L_i(X) = \mathbf{E}_{X_{(i)}} \max_j \left[\phi_{ij}(X) + L_j(X)\right]$$

The only addition was that of the $\mathbf{E}$ symbol, an operator that stands for "expected value," the average, the mean, of that which sits to its right. Under it has been placed a subscript $X_{(i)}$; by this we can say that we are averaging not over *all* of X, the entire future, but over the limited forecast given us by the seer. The other change was a formality, a substitution of the perfect information, x, with the incomplete information, X.

The power, then, of these L-equations lies in this partitioning of information. What structure is hidden away within ϕ? does every ϕ_{ij} require the whole X to make an accurate measure, or is there information about X that is simply irrelevant to some ϕ_{ij}s? Or is the power within $X_{(i)}$? are the forecasts of the future scattered throughout all of X, or are they localized, given in clusters somehow "close" to i?

Answering these questions are not easy, by any merit. Any small change to how we view ϕ or $X_{(i)}$ may change the methods of analysis entirely–much in the same vein as being allowed to reroute mid-journey changed our problem's class wholely. Nor can I make any prediction to what those answers would provide to humanity, to science, to philosophy, perhaps even art and design itself.

Herbert Simon, a notable economist who founded the idea of *satificing* to replace brute optimization and show that solving incomputable problems may still be approximatable if we are willing to except "good enough" answers, once explained everything from artistic decisions to chessplaying as a pathfinding problem through a messy, unclear space of possibilities [18]. Paradigm shifts, for any discipline, are unforeseeable in this way–satisficing feels natural, obvious, only once it has been presented to us, but discovering it for ourselves may very well be a messy, unclear journey.

But our first step–no, I cannot claim this term, "first," for mankind has fought uncertainty long since the time of Plato's Cave and before. The *next* step through this "mountain of difficulty" begins here. It begins at WKU one summer, on my bike, getting cut off course by a utility vehicle.

Chapter 2

Foundations

Our analysis of uncertainty should begin by being formally clear about the subject. I have set out to solve a problem henceforth named "safest-with-sight," which I will later resort to approximating. In this chapter I lay the groundwork of the minimum mathematics required to begin this solution. I move slowly, considering cases of the problem of increasing difficulty. To that end, this and the following chapter will be exceedingly more technical than the others. I make my best attempt to keep the language general without sacrificing rigor.

The logic ahead will expect the reader to understand themes of *graph theory*, *measure theoretic probability theory*, and *dynamic programing*. The first of these concerns itself primarily with the organized connections between entites and is commonly used to describe problems of routing and sequential decision-making. The second deals with the technical details of probabilities. Although the name contains the word "theory" twice, it can be naturally summarized by a familiar structure–a spreadsheet, one of enormous size yet built out of regular patterns–, so this topic should still be accessible to a non-mathematical reader. The last provides language for approaching problems that can be solved by solving smaller "subproblems," each solution being "cached" to reduce the total amount of work needed. The hope is that each step will boil down, down towards trivial, immediately solvable “base cases.”

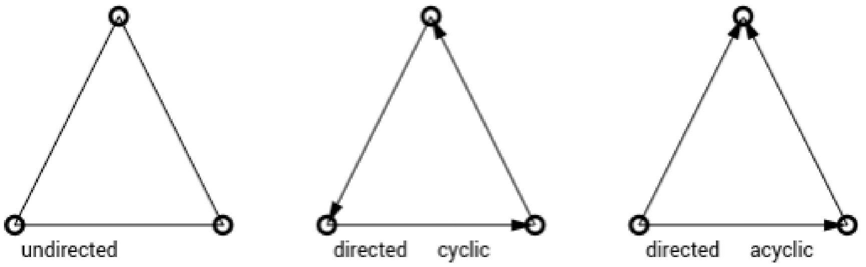
It is with these three together that we shall work towards the definition of a function P that describes, through probabilities, the behavior of an *ideal* pathfinder tasked with a simulated traversal of campus sidewalks. If we can say how the ideal pathfinder would act, why not adopt its policies for ourselves?

2.1 Graph Theory

A graph is typically written $G = (V, E)$, a *tuple* containing two components. The first, V, is a finite set of “vertex identifiers;” if i is in V, then i is the name for one of the points on our graph. Further, each point has a globally unique identifier; if there is a “Springfield,” there is only one “Springfield,” whereas “Springfield, KY” and “Springfield, CA” would represent entirely different point, as expected.

The second component, E, is a finite set of edges; if ij is in E, then there is some relationship, defined by the problem, between i and j. At this point, an important decision must be made regarding what class of graph we are using: is it *directed* or *undirected*? In the first of these, if there is a connection from i to j, then there is *not* guaranteed to be a similar connection from j back to i. An example of this would be the “parent of” connection–obviously, this relationship does not go both ways equally. In the latter class, ij’s inclusion in E *does* imply that ji is also there. For example, “is a friend of” is a mutual relationship as Facebook would have it.

If we choose the first of the two, to have directed edges, then another distinction must be made: is the graph *cyclic* or *acyclic*? In first case, there is at least one point that we can visit, depart from, and eventually find our way back to, each step moving only in accordance with the direction of the edges provided. Since a “cycle” exists, all points within that cycle may bevisited an indefinite number of times. Just keep driving in circles. Roadways, for example, are necessarily a cyclic graph. In the acyclic case, no such cycle exists.



Regardless of whether the directed graph is cyclic, a set of terms becomes necessary to speak candidly about extended relationships between vertices. First, vertex i is an ancestor of j and j is a descendant of i if there is a path, following directed edges, from i to j. Second, vertices k and h are relatives of one another if they have any ancestor in common. It is very possible for k and h to be relatives without either being an ancestor of the other; for example, consider the relationships connecting distant cousins in family trees. And third, vertices m and n are disconnected if neither is an ancestor of the other.

Next, a restriction may be placed on the graph making it *simple*, meaning that (i) at most one edge may connect each pair of vertices and (ii) no vertex connects directly to itself. Depending on how we encode a map of the country as a graph we could require either case: a map showing how many highways connect each pair of cities might require a nonsimple graph, whereas a map showing the connections at the intersection-to-intersection level does away with this need.

Finally, a frequent practice in graph theory applications, we may associate extra information with each edge and/or vertex, such as distance, cost, weight, etc. Usually these are defined through the introduction of an accompanying structure. For example D_{ij} might represent the distance between vertices i and j along the edge ij.

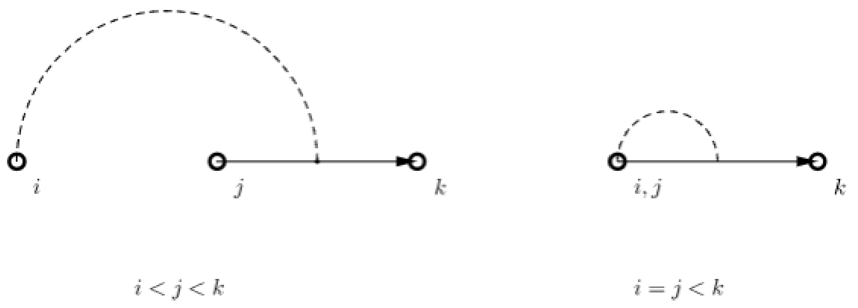
Once a suitable graph has been constructed, we can carry out common and easy-to-compute operations upon it. For example, the breadth-first-search from a starting vertex s explores all vertices that may be reached, by any path, if we were to begin at s and move forward, checking each point only once. In the undirected case, this operation would be useful for detecting whether the graph is disconnected–if it is broken cleanly into two or more "subgraphs," none having a connection outside of itself, then it is disconnected. For the directed case, a breadth-first-search can be used to determine which vertices are reachable from s–in some sense, "after" it.

A breath-first-search can also be used to construct a directed acyclic graph from an undirected one; of particular use for our problem, this will construct the graph we effectively restrict ourselves to when we follow the "cross no more than once" rule.

```
function make_directed(G, s)
    H = blank_graph()
    need_to_visit = Int[]
    already_considered = Int[]
    enqueue!(need_to_visit, s)
    push!(already_considered, s)
    while length(need_to_visit) > 0
        i = dequeue!(need_to_visit)
        add_vertex!(H, i)
        for j in neighbors(G, i)
            add_edge!(H, i, j)
            if j not in already_considered
                enqueue!(need_to_visit, j)
                push!(already_considered, j)
            end
        end
    end
end
```

This algorithm proceeds as the conventional breadth-first-search, except it adds visited vertices and crossed edges as they are discovered to a new, initially blank, graph H. It functions by keeping track of all vertices that still need to be "visited" and those that have already been "considered" for visitation. Next, it populates these lists by inserting s, our starting point, into them. Finally, until it has no more points that need to be visited, it removes a point-to-be-visited from our list, informs H that it has a vertex by that name, record each edge *incident to* that vertex, and adds all unconsidered neighbors of that vertex to the queue of vertices to be visited during future iterations. An important distinction must be made in these instructions to ensure the search is *breadth*-first: the vertex removed from the `need_to_visit` list must be the vertex that has "waited the longest." In other words, this list is a "first in, first out" structure, no different than standing in line to buy some chop sticks.

This algorithm will properly encode a map of campus as a directed acyclic graph, which captures the "cross no more than once" rule "for free." However, solving safest-with-sight still requires that a third component be considered.



So now, $G = (V, E, W)$, where V and E are as before and W stores the "lines of sight" of each vertex; if ijk is in W, then vertex i provides vision of edge jk; it is possible for i and j to be the same vertex, meaning that i can see the path to its immediate neighbor, k. It has been mentioned already that our graph is directed in safest-with-sight, as well as simple and acyclic, so integer identifiers (names like $1, 2, \dots$) may be given to the vertices; if ij is in E, then $i < j$; and if ijk is in W, then $i \leq j < k$. We also must say we have been given or somehow estimated the additional information β about the problem. This structure encodes the probabilities of each edge's "failure," which is any event that prevents the pathfinder from successfully crossing that edge; in this regard, we say that β_{ij} is the "risk" of edge ij.

I have chosen this definition of G and related graph theory concepts minimally, without sacrificing rigor or generality, while maintaining a structure that permits the use of language close to safest-with-sight's inspiration–campus sidewalks. But, it should come of no surprise that the sudden introduction of probabilities (β) leads to the requirement of additional mathematical tools.

2.2 Probability Theory

Much of the probability work that we will be doing involves the concept of a measure, which is a function that simply tells us how large some input is. It is a function μ that assigns (i) a value of zero to $\emptyset$, the "empty set," (ii) a value of *at least* zero to everything else, and (iii) assigns the value of $\mu(A)+\mu(B)-\mu(A\cap B)$ to the expression $\mu(A \cup B)$; it is much to this latter note that μ is called a measure, for we can say intuitively, that the measure of A *union* B must be the same as the measure of A, plus the measure of B, minus anything they might have in common. This "cancels out" anything that might have been counted twice.

For example, if μ is the "people counting" measure, it will count the number of people in the world that match its input. So, say T is the set of people tall enough to ride the scariest ride at the nearest amusement park to my thesis advisor, wherever he is at this very moment. Next, say R is the set of people wearing the color red.

- $\mu(T)$ is the number of tall people;
- $\mu(R)$ is the number of people wearing red;
- $\mu(T \cap R)$ is the number of people both tall and wearing red; and
- $\mu(T \cup R)$ is the number of people either tall, wearing red, or both.

Calculating this last value, if we knew the first three, is cake: the number of tall-or-red people is the number of tall people, plus the number of red people, but don't forget to subtract out the tall-and-red people, as we would otherwise count the guy leading the line and wrapped-up in a red raincoat twice.

When speaking of probability, it is natural to do so in terms of measures. Convention has it that, instead of using μ to notate our measure, we should prefer the letter P. Additionally, to compliment the "empty" value $\emptyset$, we have the "universe" or "anything" value Ω, for which we say $P(\Omega)$ is a hundred percent. Regardless of what happens, *something* in the universe will happen.

So, with P, instead of counting the number of people in the world, it counts the *percent* of people in the world. Returning to our tall-red example, $P(T)$, the probability that a tall person is randomly abducted by aliens, could be written as $\mu(T)/\mu(\Omega)$–the number of tall people divided by the number of all people, a simple fraction. Measure theoretic probability theory is, in this way, little more than counting. However, for the sake of expressiveness, probability theory comes with additional symbols, many with multiple equivalent interpretations.

First, it is common practice to abbreviate $P(\Omega - A)$, the probability of anything *but* A occuring, as $P(\neg A)$ or $P(A^c)$ or $1 - P(A)$ or $P(\bar{A})$, read "the probability of not A" or "A's compliment."

More notably, it is also common to abbreviate $P(A \cap B)$ as $P(A, B)$, read, "the probability of both A and B." Sometimes, though, the number of *events* is easier to enumerate, such as in $P(X_1, X_2, \ldots, X_n)$. In this case, for compression, we may write it $P(\{X_i\}_{i=1}^n)$. Through this use of $\{a_b\}_{b=c}^d$, we'd mean that one could "join together" with commas all the values a_c, a_{c+1}, and so on, up to, including, a_d. In the context of P, we'd mean "the probability of every single a_b happening."

Next, just as terminology exists to distinguish the extended relationships between vertices in directed graphs, probability theorists have a language their own used to maintain rigor. If A and B are mutually exclusive, then the event that A occurs nullifies any probability of B occuring, and vice versa. For example, if I break your knees tonight, then any chance of your running that race tomorrow is immediately thrown out. If A and B are joint, then knowing whether A is true has an effect on the probability of B. For example, due to the uneven distribution of wealth and education, if the aliens only abduct from Manhattan, then there will be an effect on the probability that they end up probing a Agriculture student. In constrast, A and B are said to be disjoint when the two have absolutely no effect on one another; selecting only people of a particular gender has no effect on the probability that a wealthy person is abducted. Right?

Finally, one of the most important tools in a probability theorist's belt, the concept of *conditional probability* is written in a manner like $P(X \mid Y)$, read "the conditional probability of X with respects to Y," or, briefer, "the probability of X *given* Y." This expression represents the value we would get if we restrict Ω, our universe, to only those cases where Y was true. For example, $P(R \mid T)$ would stand for the percent of tall people who are also wearing red.

Commonly, conditional probability is substituted with any of the following alternative forms.

$$P(X \mid Y) = \frac{P(X \cap Y)}{P(Y)}$$

$$P(X \mid Y) = \frac{P(X) \times P(Y \mid X)}{P(Y)}$$

$$P(X \mid Y) = \frac{P(X \cap Y)}{\sum_i P(Z_i \cap Y)}, Z \text{ are mutually exclusive and } \bigcup_i Z_i = \Omega$$

The first and the simplest is taken from how one might think about "the percent of tall people who are also wearing red." This is obviously the fraction of the number of tall-and-red people over the number of red people in general.

The second is known as Bayes Theorem. It is named after its founder, Thomas Bayes, which replaces the "tall and red" in the numerator with an equivalent expression. The replacement stems from the fact that $P(X \cap Y) = P(X) \times P(Y \mid X)$.

The last interpretation of conditional probability is the most complex, but deriving it parallels that of Bayes Theorem: instead of replacing the numerator with an equivalent expansion, we do the same for the demoninator. This expansion is that of *total probability*, a theorem that states that the probability of Y happening is equal to the probability of Y or any other one thing happening; some careful restrictions must be placed on the meaning of “other one thing.” If the “other things” that we consider are mutually exclusive with one another and, when combined together, contain all possible events within Ω, the “anything” value–that is to say, the universe has been divided into clean Z “chunks”–, then the probability of Y is the probability of Y-and-Z_1, plus that of Y-and-Z_2, and so on.

Each interpretation has its uses: the first is the easiest to state; the second is simple to transform algebraically; and the last captures closest the computational difficulties in calculating joint probabilities.

Making sense of these tools, regretfully, can at times feel unintuitive, especially when speaking abstractly with events like A, B, X, and Y. Therefore, any application of probability theory requires that the events and outcomes being reasoned about are clearly defined.

2.3 Avoiding Digital Monsters

Imagine a virtual world. This world is model of campus. The sidewalks in this simulation are connected in the same way, they have the same risks, and here we are faced by the same “game rules:”

- at each intersection,
- we can see which sidewalks are obstructed in some way,
- after which we decide which sidewalk to take next,
- restricted in that we can never visit the same sidewalk more than once.

What if we randomly set obstacles throughout the campus in accordance with the edges’ given probabilities of failure? We could then imagine a virtual version of the ideal pathfinder–it is a perfect algorithm, one that always suggests the step that maximizes its chance of reaching its destination on the other side of campus, for what it knows at the moment and might learn along the way.

Let this pathfinder loose upon the virtual world: it would still not always succeed. There is always the possibility that all entrances to the destination have been cut off, yet the pathfinder would not be informed of this disaster until it is too late to reroute around it. The oracle, although she bestows upon us information for free, she does so only in strict accordance with the lines-of-sight W.

Even in the face of the pathfinder's odds, it behaves ideally, and we could virtually follow along, recording its travels–if the pathfinder is Virgil, call us Dante, and pray that safely do we reach the end of the Inferno.

But one "trial" does not tell us everything about what the pathfinder would do. So, after the first trial, however it ended, we randomly assign new virtual obstacles and start again. After the second trial, a third, then a fourth, *ad infinitum*.

Once we have a limitless amount of data, we return to the real world. We gather what information we can–call it ξ–via lines-of-sight then measure the percent of our records where ξ occured and the virtual pathfinder took this step or that step. Ideally, by cross referencing enough information from the simulations with what we can see in the real world, we will know, with absolute certainty, how the pathfinder would behave in our shoes.

To use this probability measure, P, to this end we must decide what sort of information we would be recording on our trips through the Inferno.

We could say, for each edge ij, the event that it was obstructed is represented by α_{ij}. Conveniently, $P(\alpha_{ij})$, the probability of each failure event, is given by β_{ij}, the risk of each edge: $P(\alpha_{ij}) = \beta_{ij}$. I hope this draws an important distinction between an *event*, the thing that we are counting, and its *likelihood*, the count (as a percent) we come up with for that thing.

Next, we could say that the pathfinder decided upon crossing edge ij is represented by δ_{ij}. However, this does not, by any means, imply that made it to the other end. Imagine a scenario where the campus contains a sidewalk that is not well lit, so at no point are we given vision of it. This sidewalk is our only option left–all the other routes are visible and blocked–, so we select it. But, unfortunately, an obstacle lied there too, so our trial ends in defeat.

Third, we could say that the pathfinder reached the destination is represented by π–everyone likes pie.

Finally, to compliment each of these events, we can say by $\bar{\alpha}_{ij}$, $\bar{\delta}_{ij}$, and $\bar{\pi}$ that ij was *not* obstructed, the pathfinder chose *not* to cross ij–perhaps it never was in a position to make that choice anyway–, and that it *failed* to reach the destination, respectively.

With these in hand, we can write expressions like $P(\delta_{ij}, \bar{\alpha}_{ij})$ compactly, this one representing that the pathfinder crossed ij and did so successfully.

Consider what the following represents.

$$\sum_j P(\delta_{jd}, \bar{\alpha}_{jd})$$

If we continue our convention of letting d strictly represent the destination, then this expression is the total probability of crossing *any* edge leading into the destination, and doing so successfully. Since this can only appear in our records when we have "followed the rules" all the way up to j, this is the probability that we succeed in our trial, however we end up going about it.

$$\sum_j P(\delta_{jd}, \bar{\alpha}_{jd}) = P(\pi)$$

Therefore, since we, for the sake of mimicing the pathfinder's actions, only care about our events when they are written inside a P–that is, we only care about their likelihoods, not their particulars–, then we can discard π altogether–it is now *redundant* as far as the math is concerned.

This leaves our probabilistic model in a very minimal state:

- through α_{ij} we record the "moves" made by Lady Luck, the state of the network;
- and through δ_{ij} we record the ideal reactions to those moves.

2.4 Asking the Right Questions

As the inside-out-police-state, described in section 1.4, will attest, computer science is fraught with "alternative" thinking. Almost every problem that is solved in this field, it seems, requires first explaining something else. This "something else," before it is explained, is so remote from the original problem that any connection *couldn't possibly* be helpful. Then, as the explication comes to an end, the listener walks away just as they would from a good magician's show; a good computer science lecture leaves the audience believing two distant things are one and the same and have been all along.

I say this because, before the magician goes on stage, he must have perfected his tricks. And, no differently, before the lecturer walks into class, she must have chosen the right analogies to make, the right questions to ask, lest the proofs and solutions become muddy, correct only because "someone said so," and not obvious.

So, imagine that we have generated an infinite spreadsheet, each row recording the details of a single simulation trial. We have also written code to scour through that shreadsheet. Now all we're to do is feed it the proper input and interpret the output.

Our input could be β. Since β describes "half" of P, the $P(\alpha_{ij}) = \beta_{ij}$ "half," the values of β could be set to encode which edges have not yet been seen by leaving them at their original values and those that have been seen by setting them to a modified value. For an edge ij known to be down, we set β_{ij} to 1, a hundred percent chance of failure, and for an edge kh known to be up, we set β_{kh} to 0, no chance of failure.

And to draw a parallel with the input, the machine's output could be ϵ. Just as

β, the greek letter after α, holds the likelihoods corresponding with a set of "failed edge" events, ϵ, the greek letter after δ, would hold the likelihoods corresponding with a set of "chosen edge" events.

$$P(\delta_{ij}) = \epsilon_{ij}$$

Then, our algorithm would generally be:

1. For every edge ij we've seen or can currently see, if that edge is down, let $\beta_{ij} = 1$, or if that edge is up, let $\beta_{ij} = 0$, or otherwise just leave β_{ij} at its original value. By doing this we modify β to encode the knowledge we have collected so far.
2. For every edge ij we've selected to cross so far, let $\epsilon_{ij} = 1$, or otherwise leave it undefined for now. By doing this we initialize ϵ to encode the choices we've made so far.
3. Input the modified β and ϵ into the "machine." This is an imaginary machine, but it will suffice to illustrate our intentions here.
4. The machine will load all rows from the "infinite" spreadsheet that "match" the given β and ϵ. By doing this we are enumerating all possible scenarios that our ideal and virtual pathfinder experienced that start with the same events as we, in real life, have experienced.
5. For every edge ij where ϵ_{ij} was left undefined, the machine will let ϵ_{ij} be the *percent* of the loaded rows where ij was selected by the pathfinder.
6. Finally, the machine outputs the modified ϵ. By doing this, it "fills in the blanks" left by us in step 1.

On the surface then, this algorithm is outputing how likely it is that the pathfinder will choose to cross each edge under the given unknowns/downs/ups scenario. In some applications, this could be vital information. For example:

A social engineer is planning changes to community pathways. Numbers have been collected by the company statistician on the likelihood of each road being obstructed according to historical data. Without knowing what paths the city folk have traveled so far within a day, the engineer only knows, for certain, that the travellers will know the information about the city that the engineer has told them. See, the changes she is planning are where to place overhead road signs, which will be connected to a computer network to pass along information about wrecks and delays ahead. Therefore, she needs to know, for an ideal pathfinder, what options would tend to be made in response to information; with this in hand, she could perform simulations to determine whether additional byways would need to be added so as avoid further congestion-related accidents.

Or, another example:

A policy expert is considering what information citizens have about how to complete a particular sort of paperwork. There is an online option, a number of clerks willing to handle the transaction over the phone, and two or three centers, depending on the weekday, that will process forms sent the old fashioned way through the post office. There are several steps that need to be carried out–making sure the right signatures have been collected, the right tax information is included, approval has been given by employers, and so on. The policy expert has draw a flow chart of the whole process–alternative routes and all. He now wants to perform, for a preliminary study, a probabilistic analysis on which routes will tend to be taken, depending on at what steps certain information is made available to the participants: information such as office hours, scheduled site maintainance, and other “downtime” during which going a different route would be quicker.

These examples are related to safest-with-sight, but only marginally. Still, they are asking *precisely* the right questions if we take the algorithm's "modifed ϵ" output at face value.

"Dig deeper into the meaning of ϵ," the good lecturer says. Consider what values it would contain if we give "perfect" information–if we tell it through the values of β everything there is to know about the ups and downs of the network. Because the pathfinder is ideal, it does not behave randomly, so if we expose it to the same scenario repeatedly, it will respond identically each time. Therefore, ϵ will contain only 1s and 0s, values effectively letting us know "yes" and "no" for the question, "Is this what the pathfinder would do?"

In practice, though, we would not have the "true" values of β. If we did, then we needn't worry about this problem at all–the oracle has told us everything, so we can take any path we see that reaches the desired destination.

As long as β contains 1s or 0s for everything the pathfinder could see in the simulation at the point in question, then it's behavior will still be deterministic. This is because all other information is of no use to the spreadsheet-scouring algorithm: adding any information to β, other than what is required for determinism, can have no effect on ϵ because, since it was unknown to the pathfinder during the trials, it can have no effect on δ, the events capturing the ideal behavior.

This concept deserves a name. If the pathfinder were truly optimal, then it would always find a path to the destination when any such path exists. But, this can only be done when the oracle tells us everything at the onset. More realistically, then, the pathfinder is *sight-wise optimal.* It behaves in whatever way maximizes its chance of success, according only to what it can see, has seen so far, and knows about the probabilities of the other edges

This term gives further insight into the task at hand, to interpret the output of this algorithm. Although asking the question, "Is this what the pathfinder would do?" accomplishes the goals of safest-with-sight, to describe how one should behave in the face of partially-visable uncertainty, we can do better.

Because the pathfinder is sight-wise optimal, we know that it always selects the edge that maximizes the probability of successfully reaching the destination. Therefore, a more general question, one that even the pathfinder itself must ask is, "How likely will I succeed if I choose this edge next?" Given the answer to this for each edge that could be selected next, we go the way of the ideal and choose whichever choice had the highest "score."

This, I hold, is the *right* question to ask. The first two interpretations of ϵ provide interesting information in their own right, and they are both direct derivatives of this hypothetical algorithm, but this last sort of query represents the true nature of the algorithm–it represents not the input, nor the output, but a means by which that output might be decided.

Our final step, then, is to tackle this new perspective on the safest-with-sight problem, a task which we will do with the language of dynamic programming.

2.5 Dynamic Programming

I hate the name Dynamic Programming.

The coiner of the term, Richard Bellman, admitted that the term was purposely selected to be distracting in the 1950s politics surrounding his research [5]. The true nature of dynamic programming is that it is a mathematics for planning–a common vein of the computer science field. The general flow is little more than the following:

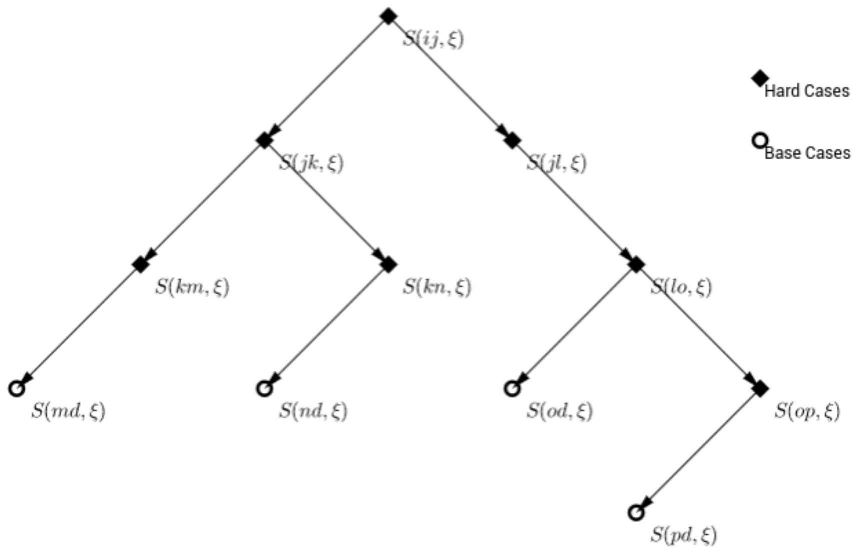
1. Begin with a problem.
2. If that problem is too hard, solve a simpler "subproblem," or even multiple if need be.
3. Combine the results of the subproblems to find a solution for this problem.

4. Keep the answer to this problem written down in case you need to solve it again.

This paradigm is a good fit for problems whose solutions can be expressed as some summary of the "future," such as the maximum, minimum, average, etc. of those futures. When we can write the solution, say to problem i as a function of its *subproblems*' solutions, say the solution to each j "next to" i, we should thank Dr. Bellman.

$$S_i = f_i(\{S_j\}_j)$$

The only other requirement for the use of dynamic programming is the reduction to *base cases*, to subproblems so simple that their solutions are trivial. The time it would take to solve the desired problem, then, would be proportional to the number of subproblems that we needed to first solve–that is, the *space* filled by the cache of subproblem solutions. The time it takes to paint a wall, after all, is always relative to the size of the barn.



Following the instructions of a dynamic program, the machine would begin at the "top," at the hardest problem to solve, then work its way down, towards successively simpler cases. Eventually, it reaches a base case, whose solution becomes the first entered into the cache. It may then solve and store other base cases until it has enough information to solve and store a subproblem–a subproblem still much simpler than the original, one just above the base cases.

And so on, it continues, solving solving harder and harder subproblems. As it nears completion much of the work required by the harder subproblems near the "top" may have already been entered into the cache, so whole "trips" back down needn't be made twice.

This "automated simplicity," so to say, lends itself to (i) few required lines of code and (ii) straightforward formal proof that the code is, indeed, correct. These, of the latter point here, take the form of a *proof by induction*:

1. Prove the succession of subproblems always approaches a base case. If we will never stop breaking the problem down into subproblems, then we will never finish solving it.
2. Prove all base cases are correctly solved. If any one of the base cases are handled improperly, the entire logic of our solution collapses.
3. Prove any non-base case will be correctly solved by assuming all of its subproblems will be solved correctly. With this and the previous two points shown, the proof, in a sense, completes itself, built, just as our solution is, recursively from trivial base cases.

This will be the general outline of our treatment of safest-with-sight, calling back to concepts of graph theory and probability where necessary. Before I proceed with the analysis, though, we must first make clear, just as we made clear the events recorded within our probability outcomes, how we plan to frame safest-with-sight in a dynamic programming context.

First, it is precisely meant, by a solution to a safest-with-sight problem, the probability a sight-wise optimal pathfinder would successfully reach the destination if an edge ij is taken, given that it is currently situated at vertex i and it currently knows some information ξ about the status of the network. From this perspective, a problem is defined entirely by the two inputs ij and ξ. Therefore, we can say what we are seeking an exact means to calculate in a much more compact expression.

$$S(ij, \xi) = P(\pi \mid \delta_{ij}, \xi)$$

Next, a subproblem for helping find $S(ij, \xi)$ can be denoted $S(jk, \xi')$. Here, jk is obviously any possible step that would be available after ij and ξ' is what information will be available to us in the future–what we know now plus what we will learn upon reaching j. This latter variable, ξ', is a slippery creature, for although we do not know what information we will know in the future, we still know what edges we will have information about. The following will become a useful gear for our probability machine.

$$P(\xi' \mid \xi)$$

We may frequently be interested in the likelihood of some possible future occuring; however, we mustn't forget to account for what is already known, mustn't forget to use the conditional probability with respects to ξ.

Finally, I must attend to the base case, the elements that will compose our final solution. As we make steps successively closer towards the destination, at some point we will reach that destination and be done with it. But move back *one step*, to that *last step*. We are guaranteed that there is at least one step that is closer to the destination than any other along its path; there may, in fact, be several such sidewalks coming into the destination from alternative routes. Solving this last-step subproblem, $S(id, \xi)$, is none-so-brag-worthy, as their solutions had already been given to us–in β.

Chapter 3

Analysis

There is only one step left. Our destination is just outside our reach. We've come far, far enough that there is no way out–we are committed. This is the base case, speaking dynamical-programming-wise.

3.1 As Easy As Possible

The solution for this case is denoted $S(id, \xi)$, where d is understood to be the destination and id is understood to be that last step. Since the solution to a subproblem is the probability that we will reach our destination, we can state the value of S here quite easily: if the edge is down, we fail, and S here is 0; if the edge is up, we succeed, and S here is 1; if we have no idea whether id is a failed edge, then S here can considered the probability that id is in the more favorable state.

We do not have the probabilities on hand for the probabilities of each edge's succeeding, not directly; however, we do have the probabilities of each edge's *failing*, β. Success is just the compliment of failure, so the base case solution should be obvious.

$$S(id, \xi) = 1 - \beta_{id}$$

This is incomplete, only accurate when the edge's state is still unknown. The information of what edges have been seen so far and their states are "encoded" within ξ. When ξ contains α_{ij}, we know ij has been seen and that it has failed; when ξ contains the compliment, $\bar{\alpha}_{ij}$, we know ij has been seen and that it is intact; otherwise, ξ contains neither $\alpha/\bar{\alpha}$ term and the above solution holds. Therefore, we can give a more verbose formulation of S here.

$$S(id, \xi) = \begin{cases} 1 & \bar{\alpha}_{id} \in \xi \\ 0 & \alpha_{id} \in \xi \\ 1 - \beta_{id} & \text{Otherwise} \end{cases}$$

Although we've given the math expression of the base solution, it is always important to consider how this will be implemented by a machine, the algorithm, the unlucky intern. Since the solution is one of three fixed values, we can use a three way logical branch.

```
function baseCase(beta, xi, i, d)
    if edgeKnownAndUp(beta, xi, i, d)
        return 1
    elseif edgeKnownAndDown(beta, xi, i, d)
        return 0
    else # edge unknown
        return 1 - risk(beta, i, d)
    end
end
```

The runtime required for this algorithm relies on the runtimes of the methods that were assumed, `edgeKnownAndUp`, `edgeKnownAndDown`, and `risk`. How these operate are in turn contingent on the structures the machine uses to store β, α, ξ, and so on. Some approaches to these storage mechanism could be efficient in terms of space–bits of memory required–, but suffer with a trade off in time–steps required to perform basic operations; for others, the concern would be reverse. Nonetheless, I do not fear that selecting an appropriate structure would be an arduous task, even for the junior programmer.

For example, as I have elected to use my own simulations, one may use a *sparse matrix* for many of these structures, in particular a *sparse dictionary of keys.* This is a structure that stores data and allows insersions and look ups to be carried out almost immediately. It's power, though, comes in how it treats missing values–those we have not yet inserted. Instead of raising an error and crashing the program as a non-sparse structure would do, it responds with whatever value is assumed to fill all the "empty" space of the structure–thus the name *sparse.*

Regardless of how the data is stored, it will have no effect on the simplicity of the base case solution–we have three cases to check and we respond with whatever fixed value is appropriate. And, even when we step back from the destination further, the subproblem to be solved can still be just as trivial to solve.

Consider the case where we've step into not a final step, but a final "leg" of the journey. From here on out, there are no more choices to be made. The path ahead no longer branches, there are no "lefts" or "rights" to choose from. We just keep moving. It doesn't matter in this "final leg" case what we see along the way: our only chance of success is if *all* the edges ahead of us succeed. If any one of them is down, the entire path is worthless and we fail the trial.

To this end, it may be useful to have a form of the base case calculation that can be used at any time, on any edge.

$$\hat{S}(ij,\xi) = \begin{cases} 1 & \bar{\alpha}_{ij} \in \xi \\ 0 & \alpha_{ij} \in \xi \\ 1-\beta_{ij} & \text{Otherwise} \end{cases}$$

Here, $\hat{S}$, read "S hat," does exactly what S does in the base case–it gives us the probability of that one edge being up, as far as we can tell via ξ–, but for any edge at all. With this, solving the "no more branches" case becomes straightforward. Imagine that we are at vertex i. Ahead of us, in a straight line, are j, k, and so on, ending with d; our path ahead will be something like ij, then jk, and so on, ending with zd. The exact number of steps does not matter, just so long as the traversal is without branches.

From what we can see, right now, from i, according to ξ, the variable that holds all the information we've learned so far, we can only succeed if ij is up, then jk is up, and so on.

$$S(ij, \xi) = \hat{S}(ij, \xi) \times \hat{S}(jk, \xi) \times \cdots \times \hat{S}(zd, \xi)$$

If any one of these calls to $\hat{S}$ returns zero, meaning that any one of these edges are down, then the whole multiplication result will be zero–the whole path might as well have failed.

It doesn't matter if the probabilty of success will change as we continue to walk along that path and learn more. Obviously, everytime we see that more of our path is unobstructed, we can rejoice as S increases. That is not what S, the solution to a subproblem, is asking.

A solution to a safest-with-sight problem is not the path itself. It only cares about (i) what is known *right now* and (ii) what is known *at decision times*. Because a straight path provides no such case of the latter, only the present moment's knowledge, that of i, matters for the current moment's subproblem solution, that of the probability of success from i through j and on towards the destination.

With this last point in mind, then, it should come with no surprise that adding branches–opportunities for decisions–changes how solutions must be approached and formulated altogether.

3.2 Like the Opposite of Easy

This was introduced earlier as the total probability formulation of conditional probability. Here, we begin to use it.

$$P(X \mid Y) = \frac{P(X \cap Y)}{\sum_i P(Z_i \cap Y)}, Z \text{ are mutually exclusive and } \bigcup_i Z_i = \Omega$$

A fair warning, syllogism follows. To that end, keep what we are moving towards in mind: an expression, as detailed as necessary to convert that expression into executable code that calculates *the probability of eventually reaching vertex d, given ξ, assuming we are at vertex i and will move along edge ij next.*

That is, find x, where $P(\pi \mid \delta_{ij}, \xi) = S(ij, \xi) = x$.

Now break x in two: $x = x_1 \times x_2$. We are using $\times$ to represent typical multiplication, preferring it here in an aspect of readability.

Let x_1 be the probability of success *now* and x_2 that of *later*.

When we cross ij, one of two things will happen. The first, we could fail immediately–the "now," or x_1. The second, we could succeed in crossing ij, then continue–the "later," or x_2.

In the same vein as solving the "no branches" case, it is obvious that $x_1 = \hat{S}(ij, \xi)$.

That "later" case can then assume that we crossed ij successfully. Other than that, though, the "later" case is complicated. Our probability of "later" success is dependant on what we learn once we reach j. However, we have not yet learned that information. So, we can imagine *all possible cases* of what could be learned and, following total probability, combine them into a single expression.

$$x_2 = \sum_{\xi'} \left[P(\xi' \mid \xi) \max_{jk} \left[S(jk, \xi') \right] \right]$$

Here, $\sum$ echoes the sum of mutually exclusive Z_i events above, so if ξ' is just *one* of the *all possible cases* of what could be learned upon reaching j, then $P(\xi' \mid$

ξ) is the likelihood of that case happening. We must remember to include within that likelihood the condition that everything in ξ is already known–we only want to consider the likelihood of *new* information.

So, x_2 can be found by enumerating all possible values of ξ'. For each, we find the max $S(jk, \xi')$, which is the probability of success of the best next step, the one the ideal pathfinder will certainty take. Therefore, x_2 is the *expected value* of the *best subproblem's* solution.

$$x_2 = \mathbf{E}_{\xi'} \left[\max_{jk} \left[S(jk, \xi') \right] \right]$$

Reunite the xs, then with minimal algebraic rearrangement, arrive at the solution.

$$S(ij, \xi) = x_1 \times x_2$$

$$S(ij, \xi) = \hat{S}(ij, \xi) \times \mathbf{E}_{\xi'} \left[\max_{jk} \left[S(jk, \xi') \right] \right]$$

$$S(ij, \xi) = \hat{S}(ij, \xi) \times \sum_{\xi'} \left[P(\xi' \mid \xi) \times \max_{jk} \left[S(jk, \xi') \right] \right]$$

$$S(ij, \xi) = \sum_{\xi'} \left[\hat{S}(ij, \xi) \times P(\xi' \mid \xi) \times \max_{jk} \left[S(jk, \xi') \right] \right]$$

$$S(ij, \xi) = \sum_{\xi'} \left[P(\xi' \mid \xi) \times \max_{jk} \left[\hat{S}(ij, \xi) \times S(jk, \xi') \right] \right]$$

$$S(ij, \xi) = \mathbf{E}_{\xi'} \max_{jk} \left[\hat{S}(ij, \xi) \times S(jk, \xi') \right]$$

The similarities between S here and L_i, in an earlier chapter, are worth noting, as doing so will give insight into the time required to compute their results exactly in a “less than simple” case.

Before, I defined L_i as follows.

$$L_i(X) = \mathbf{E}_{X_{(i)}} \max_j \left[\phi_{ij}(X) + L_j(X)\right]$$

In the original, L_i, i and j were understood to represent neighboring states in which a decision maker could find herself. In the parallel, ij and jk can be understood as not neighboring states, but neighboring edges–subsequent decisions. Further, $X_{(i)}$ has become ξ', although the analogy between the two can again be easily drawn: both represent a set of knowledge provided by reaching some point. Although $X_{(i)}$ represents the knowledge provided by i and ξ' represents the knowledge “accumulated” up to j, we must keep in mind how the $\mathbf{E}$ operates–it, in L_i, together with $X_{(i)}$, mimics the “accumulation” of information present more clearly in S.

What is different between the two lies in that ϕ has finally been given form–as $\hat{S}$–and that S combines values with multiplication instead of addition. Similarly, X has been defined as a form of ξ, a collection of “seen up,” “seen down,” and “not yet seen” values. Therefore, we might be able to say–with further analysis not carried out here–that calculating L_i is *at least as hard as* doing the same for S–the latter may be just *one* special case of the former, which, it is easy to imagine, contains several other special cases, some of which might require even more effort to solve than safest-with-sight.

So how hard *is* our problem? In it’s most compact formulation, it is tranquil.

$$S(ij, \xi) = \mathbf{E}_{\xi'} \max_{jk} \left[\hat{S}(ij, \xi) \times S(jk, \xi')\right]$$

But this equality conceals a sea of complexity: because the expected max operation must iterate over all possible realizations of ξ', that step alone may consume an ornate amount of computation. Imagine that, upon reaching j, we see at most h edges that have not been seen before. This captures every possible case, since h can always be selected to be large enough. The number of realizations for ξ' is 2^h, two cases, one up, one down, for each of the h edges, so $2 \times 2 \times \ldots$.

But let's assume h is a small number, maybe 2 or 3. This way, there is a limit to the amount of new information that must be anticipated by the expected maximization operation. However, this does not prevent an even larger problem from happening.

What if every single path to every single vertex i produced a different ξ? If this happens, then caching solutions will be of little help, since each time we reach i in our exploration of subproblems, all the work ahead of it must be computed again. The total number of operations, then, would be 2^v–where v is the number of vertices in between the starting point and the destination, each of which may or may not be included within whatever path we take, so again $2 \times 2 \times \ldots$. Restricting h may be an easy assumption to make–I can only see so far–, but limiting the number of paths is less than permissable. We have little or no control on what roads lie ahead of us after all.

So, it may seem that safest-with-sight cannot escape the jaws of $2 \times 2 \times \ldots$. We are all doomed, and the Sun will singe us all long before anyone can figure out what step to take first to cross campus. Why bother thinking this hard at all?

3.3 Difficulty is a Fine Line

Safest-with-sight's computational complexity requires one of two things to be occur: either h, the maximum amount of new information gained upon reaching any vertex, must be large enough; or each path through the graph must produce a different "accumulation" of knowledge. Without these, safest-with-sight falls into a category of problems known to have very, very efficient solutions.

Andrey Markov, the story goes, studied a problem of *state transitions* around the time between the nineteenth and twentieth centuries. A system M is said to have n states. For each pair of states i and j, we are given the probability that the system will transition from i to j next whenever it is in i, written M_{ij}.

In this way, M is a matrix storing the likelihoods that precisely describe how the system behaves by moving from one configuration to the next. Because these are probabilities:

- each M_{ij} is between 0 and 1; and
- the sum of each column of M totals to 1.

In other words, there are no negative probabilities or those greater than a hundred percent. Also, each state has a hundred percent chance of going *somewhere*, even if that somewhere is staying still (the state leads back into itself).

The most important feature of *markovian* systems is that they are *memoryless*. It *does not matter how we got to i*, we will still behave with the same probabilities. When safest-with-sight has two paths to the same point that produce different sets of information, we are faced with a non-markovian system; but when all paths to each vertex produce the *exact same* set of information, our behavior can be said to be memoryless and we, when we reach that vertex, will always proceed in the same manner–thus reducing the required computations by fathoms.

First, we start our solution at the end: since it doesn't matter how we get to the easiest cases, $S(jd, \xi_j)$, we know ξ_j will be the same (we give a subscript here for disambiguation for the current discourse only). So, we find ξ_j, a simple task accomplished by finding *any* path leading from the starting point to i and recording what information we will pick up along the way. Next, we calculate S here, which we know to be an easy value to produce since it is a base case. Then, we step back to i, a point that leads into j, and solve that subproblem, $S(ij, \xi_i)$. Since all possible subproblems of i will have been solved and thier solutions stored, solving this problem will be simple too.

And so, working backwards this way, we visit each edge exactly once. During each visit, 2^h realizations of ξ' may be needed, but when h is small enough this term can be forgotten. Therefore, the necessary number of computations needed is some small multiple of E, the number of edges in the graph, a feasible runtime indeed.

What we are seeking, then, as we begin to convert S to code, as we have already done for $\hat{S}$, is to design our "problem solving system" to be a "machine" that will behave efficiently when a markovian problem is given it or when it discovers a markovian subproblem hidden within a larger, non-markovian problem. This requires several components working in tandem:

- storage structures for the graph and β;
- a component for solving $\hat{S}$;
- a component for encoding, storing, and retrieving inputs and solutions to S;
- a component for iterating over possible ξ' realizations;
- and a coordinating component, for solving S.

Before we design these components, though, we believe it necessary to first consider the special cases in which the machine can perform fast and to lay out exactly how solutions in these cases should be calculated.

3.4 Slight Adjustments

Regardless of the case we are considering, we can improve our lives by making two adjustments to the structure of the vision.

First, if every parent of a vertex provides the same line of sight, then *effectively* the child of all of these parents may as well provide that same line of sight itself. Repeatedly adding such "effective lines-of-sight" to the graph (i) does not change the meaning of the graph as it pertains to our problem and (ii) reduces the complexity when considering a "working backwards" approach to solving the problem.

And second, if a vertex provides vision of an edge, but there is no path connecting the two, then why bother with that information? In these cases, we can just remove the line-of-sight altogether, again with no effect on the meaning of the problem and greatly simplifying the analysis.

Therefore, hereonout, these two adjustments will have been assumed aready made.

3.5 Minor Improvements

In the simplest possible case, there is no vision, no lines-of-sight, no 2^h to worry about. In these cases, ξ is always the same value, regardless of where we are in solving our subproblems–it is empty. Also, though, without any information to guide it, the pathfinder will be forced to select a single path and stick to it–the path that, from much earlier, minimizes the number of "special coins" that must be flipped.

In this case, our code just needs to solve the problem from the bottom-up, each step based on the maximum probability of success through each possible next step, visiting each edge only once in all. Dynamic programming, by working towards the base case, the last possible step, and storing solutions for later, automatically behaves in this way for us.

When we allow vision to be of *incident* edges only, to only allow vision from i to be of the edges to its neighbors ij, we achieve a similar speed to before, but we must be careful in the *order* that we do things. This case is obviously markovian, since each time we step, the only information brought along with us is of edges now behind us, so it can be forgotten. Therefore, we should be able to work in the same bottom-up, cache as we go manner as the "no vision at all" case. The difference, though, comes when we are combining solutions to subproblems. Instead of just taking the best next step, we have to consider the case when the best might be down, so we would resort to the second best, unless it's down too, and so on.

Let's say that we are at i and that its neighbors, the js and ks, are named $1, 2, 3, \ldots$, in order from best to worst.

$$\text{if } j < k \text{ then } S(ij, \xi) \geq \hat{S}(ik, \xi)$$

If j comes before k, then we know that our likelihood of success is better, or equal, if we choose to take ij next. Of course, we would not be normally given vertices with names in this way, but once we've calculated the value of S for each subproblem after i, *temporarily renaming* the vertices, to no effect on the meaning of the graph, allows our math to be stated much more compactly.

So, we ask, "What is our probability of success if we start at i, given ξ?" This is different from what S calculates, since it does not include the assumption that we are taking some edge ij next; having it, though, makes the calculation of the latter much simpler. Hell, this question is just the value of x_2 from before, when we reasoned that x_2 needed to be an expected maximum. When our pathfinding behavior is more regular, as it is in the markovian case, we can state it more descriptively.

$$x_2 = S(i1, \xi) + \beta_{i1} S(i2, \xi) + \beta_{i1}\beta_{i2} S(i3, \xi) + \ldots$$

When our best option is available, we will cross that edge and continue. This probability is already given–it is simply S. However, when the best option is down, which happens with probability β_{i1}, we will take our second option in the same manner, and so on, but then when do we stop this pattern?

It may seem that we continue through all possible next steps, but that is not the case–not when there is at least one option we cannot see. These options are "gambles," since the pathfinder is given no assurance that it will cross safely.

Let's say that the *best gamble*, in this sense, is named g. If we are confronted with multiple gambles, we will only ever take the best of those, and if there is a *non-gamble* whose "score" is worse than g's, we will never choose that non-gamble either–why would we? So, the last term in our sum, for x_2, must be that corresponding with g, if such a gamble exists.

$$x_2 = \sum_{j=1}^{g} S(ij, \xi) \prod_{k=1}^{j-1} \beta_{ik}$$

If we plug this modification of x_2 back into our definition for S, we receive a regretfully more complex, but easier to compute, function.

$$S(ij, \xi) = \hat{S}(ij, \xi) \sum_{k=1}^{g} S(jk, \xi) \prod_{m=1}^{k-1} \beta_{jm}$$

3.6 Planning for the Future

There is a fundamental difference between lines-of-sight that are incident and those that are not, between the "near" and the "far," between the "now" and the "later."

When we are given information that we can use now, our behavior is a simple direct response to that knowledge. We choose the best reaction to make, we make it, and we do away with that info–it will aid us no further. Even when the vertex that provides that vision is ahead of us and we have yet to learn what those lines-of-sight have to teach us, anticipating that information is simple too: we are, of course, anticipating not *just* what moves Lady Luck will make, but what moves we might make to counter. Our future responses to this fleeting information of incident lines-of-sight, in markovian systems, are trivially predictable, modeled, and reasoned about. Therefore, working through the calculations involved with this information is straightforward, linear–*easy*.

But information even *one step further* than incidence is–well, it's damn hard to deal with. The probabilities associated with paths to the left versus those on the right are *disjoint* up to incident edges; these, by the nature of directed acyclic graphs, do not "overlap." The first step I can take in either direction can never, in any case, be a step that could be taken at some point later if I were to choose the alternative path instead.

The *second steps* in either path, however, do not hold this property. Once we move past the incident lines-of-sight, our probabilities can become *joint* again, overlapping in some sense, having in common the variables that make up their values. It is because of this that, I am afraid, we must consider all 2^h possible combinations of the information we might learn at each step when vision is non-incident. Whereas the disjoint probabilities of incident edges can be reasoned about algebraically, the joint probabilities of further edges require an exhaustive search through a slew of "what ifs." It is this latter point that causes "future planning" in safest-with-sight and similar problems to take too long to calculate to be useful.

Thus, we combine the two.

$$S(ij,\xi) = \hat{S}(ij,\xi)\mathbf{E}_{\tilde{\xi}'} \sum_{k=1}^{g} S(jk,\tilde{\xi}') \prod_{m=1}^{k-1} P(\alpha_{jm} \mid \tilde{\xi}')$$

Here, $\tilde{\xi}'$, like ξ', represents *realizations* of how ξ might change upon reaching j. However, $\tilde{\xi}'$ is only a *subset* of ξ' for it does not realize the yet-unknown statuses of incident edges to j; so, if information tends to be of edges closer to a vertex, as is easy to imagine, and only $\tilde{h}$ edges' worth are non-incident, then only $2^{\tilde{h}}$ cases of $\tilde{\xi}'$ need be considered.

This new equation considers all information, but it partitions how we reason about incident and non-incident information: one as a weighted sum, one as an expected maximum. By doing this, it can be more easily converted to code that runs $2^{h-\tilde{h}}$ times faster than it would have if we naively follow the original formulation of S. S has grown large, so I have given a formal proof of its correctness in the appendices.

As one should expect though, the code for this will either be incomprehensibly compact or lousy with specialized components. I like the latter.

Chapter 4

Solution

Before we design the code for calculating the newest formulation of S, an important note needs to be made about β and ξ–they are, as far as we are computationally concerned, the same object.

Whenever an edge has been seen by the pathfinder, the corresponding α or $\bar{\alpha}$ is inserted into ξ, thus allowing the accumulation of knowledge about what edges are down or up, respectively. To a similar end, β holds the original probabilities of each edge's being down; what does it mean, then, if $\beta_{ij} = 1$ for some edge? or $\beta_{ij} = 0$? In these cases, we know that the edge is either always or never obstructed from the very beginning.

Why don't we just modify β then, leaving ξ out of the equation altogether? This is, of course, figuratively speaking. For the sake of simpler proofs, we wish to keep the two concepts distant "in the math," which allows analysis to be carried out much clearer. However, doing the same when we are coding causes more headaches than its worth.

Therefore, in the code here, instead of inserting an "edge is down" or "edge is up" token into whatever structure represents ξ, we will modify the values of β, saying, when an edge is seen to be down, that the probability of it being down is, from here forward, is a hundred percent; and similarly so when an edge is seen to be up. These modifications will be unmade, the corresponding values of β returned to their original values, whenever we are done with that information and need to "move back up" to an earlier subproblem when that edge was out of sight.

For example, we can now state the base case, when only a single edge is being considered, in a single line of code.

```
function baseCase(beta, i, d)
    return 1 - risk(beta, i, d)
end
```

When id is down, β_{id} equals 1, so the probability of success is $1 - 1$, or simply 0, impossible. Conversely, when id is up, β_{id} equals 0, so the probability of success is $1 - 0$, or simply 1, guaranteed. In all other cases, when the edge has not been seen, the probability of success is the inverse that edge's risk, or $1 - \beta_{id}$.

On a related note, the general case can be written in only a few lines as well.

```
function generalCase(G, beta, i, j)
    key = encodeInput(G, beta, i, j)
    if not haskey(CACHE, key)
        if j == DESTINATION
            CACHE[key] = baseCase(beta, i, j)
        else
            CACHE[key] = baseCase(beta, i, j) * expectedMaximum(G, beta, j)
        end
    end
    return CACHE[key]
end
```

In fact, only a single line is doing any "real" work, that where the solution's value is being inserted into the cache. The complexity of the problem is then masked away behind the calls to two "helper" functions, `encodeInput` and `expectedMaximum`.

4.1 Encoding

Imagine that we are faced with two subproblems. Apart from only a single edge, the two are exactly the same: in one, the edge has been seen and is up; in the other, the edge has been seen and is down. Are these two in fact the same?

The answer to this depends on where that edge is. What if we can, from our current position within the graph, still reach that edge? Perhaps it is behind us, or perhaps it has been cut off by a “wall” of downed edges. If that edge is still visitable, then it may be best to consider these two subproblems different and solve them individually. But otherwise, when we are disconnected from the edge, then who cares? We can just treat them as the *same* subproblem. They are, after all, effectively the same thing.

This idea is central to how I have chosen to encode input for our subproblems. For each input, we produce a sequence of 1s, 0s, and question marks. Every 1 corresponds with an edge that is seen and up; 0, seen and down; and question mark, unseen. However, we also use a 0 to represent an edge that is unvisitable–as far as I care, it might as well be down.

```
function encodeInput(G, beta, i, j)
    encoding = str(i, "/", j, "/")
    if edgeUp(beta, i, j)
        encoding += "1"
    elseif edgeDown(beta, i, j)
        encoding += "0"
    else
        encoding += "?"
    end
    if not edgeDown(beta, i, j)
        for (m, n) in bfsearch(g, j)
            if connected(G, beta, j, m, n)
                if edgeUp(beta, m, n)
                    encoding += "1"
                elseif edgeDown(beta, m, n)
                    encoding += "0"
                else
```

```
                    encoding += "?"
                end
            else
                encoding += "0"
            end
        end
    end
    return encoding
end
```

The code for this encoding process appears long, but its length is due only to the multiple times we have to consider "if this then that, or otherwise this," which correspond with the different points at which we must decide what to add to the encoding: a 1, a 0, or a question mark. To cut out ambiguities, we should start each code with a label of what vertex we are at, i, and what vertex we are considering visiting next, j. Finally, the order of the ones and zeros and unknowns corresponds to a breadth-first search that starts at j. Since the breadth-first search only visits edges that j is *possibly* connected to, it will automatically ignore the edges that are behind j. It still must check, however, if we are *actually* connected each edge. This is to to account for the cases where an edge ahead of j might be cut off from it by edge-failures.

Once we have this encoding, we will know where in a dictionary-like structure to look if a solution has already been added. Much like an English dictionary, a data-dictionary allows the computer to quickly locate the value of an entry corresponding with the name of the entry; the encoding here is that name. It is the job of the `generalCase` code to calculate the solution's value for the first time and write it into the dictionary for later look-up. Although this may require a large amount of memory storage to record all the necessary values, the time saved is worth the tradeoff.

4.2 Expected Maximum

With additional helper functions, the `expectedMaximum` too can be written simply. We can use a `realizeFuture!` function–the `!` is a Julia convention to note that it modifies at least one of its inputs, in this case `beta`–to iterate through all the possible realizations of the *nonincident* edges that will be seen upon reaching the next vertex. We can then calculate the expected maximium by a sort of weighted sum.

```
function expectedMaximum(G, beta, nextVertexToTake)
    answer = 0
    for likelihood in @task realizeFuture!(G, beta, nextVertexToTake)
        answer += likelihood*incidentAverage(G, beta, nextVertexToTake)
    end
    return answer
end
```

This works through `realizeFuture!`'s returning, for each realization in the iteration, the likelihood that that realization occured. Just before this likelihood is returned, the value of `beta` is updated to reflect the realization.

```
function realizeFuture!(G, beta, s)
    found = (-1, -1)
    for (i, j) in nonIncidentVision(G, s)
        if edgeUnknown(beta, i, j)
            if connected(G, beta, s, i, j)
                found = (i, j)
                break
            end
        end
    end
    if found == (-1, -1)
        produce(1.0)
    else
        i, j = found
        temp = risk(beta, i, j)
        makeEdgeUp!(beta, i, j)
        for likelihood in @task realizeFuture!(G, beta, s)
            produce(likelihood*(1-temp))
        end
        makeEdgeDown!(beta, i, j)
        for likelihood in @task realizeFuture!(G, beta, s)
            produce(likelihood * temp)
        end
        makeEdgeUnknown!(beta, i, j, temp)
    end
end
```

To minimize the number of lines of `realizeFuture!`, I have defined it recursively. When it is first called, from within `expectedMaximum`, it searches for any edge that meets the proper criteria: any edge that is within the nonincident vision of our next step, that is yet-unknown, and that is our next step is connected by some path, we consider the `found` edge. If we are unable to locate such an edge, then we do not need to update `beta` and we just *produce* the likelihood `1.0`, a hundred percent.

The `produce` function–whose name comes from the classic producer/consumer problem in distributed programming texts–, in Julia, along with the `@task` decorator is used to create functions that can return a value to an iteration elsewhere, pausing their own execution. After each step of the loop, the paused function resumes, continuing exactly where it left off until it either produces another value for the elsewhere-loop or ends. In the latter case, the loop has no more values to *consume* and halts.

The reason I have chosen to take the `produce` approach to writing `realizeFuture!` is made clearer when we consider what happens when it *does* locate a `found` edge. In this case, it must modify `beta`. This is broken into five stages:

1. Update the value of `beta` such that the found edge is up. This does not mean that we know, right now, from where we stand at the entrance to campus, that the edge is up. We are instead just consider one "what if" that might happen.
2. With the found edge up, we loop through `realizeFuture!` again, which will find a different edge, modify it, call `realizeFuture!` again, find another edge, and so on, until the base case of the recursion is reached: the case when no

more edges can be found. In this way, the first realization considered is the "optimistic case," where all edges that we will see will be seen up.

3. For each realization we see here, we re-produce the `likelihood` we recieve from the recursive call, but we first multiply that likelihood by the probability that the found edge would actually be up.
4. After considering all cases where the current found edge is up, we change the value of `beta` again. This time, we set it to where the found edge is down. Then, we recurse again, exploring all the what ifs of all the other edges that will be found again. In this case, we re-produce likelihoods just as in the previous step, except the multiplier is the probability that the found edge would actually be down, not up.
5. And finally, after all recursion is complete, we return `beta` to its original state, ending the method and whatever elsewhere-loop that initiated this call to `realizeFuture!`.

In this way, `realizeFuture!` modifies `beta` to every possible value that it might be, calculating the likelihood for each state.

The last helping function we need to define is the `incidentAverage`, which, in a much simpler process, considers the up/downs of incident edges. These were previously ignored by `realizeFuture!`, which adopts the `nonIncidentVision` helper, because, as discovered during analysis, we can process incident edges more efficiently by using a different technique.

```
function incidentAverage(G, beta, i)
    options = Any[]
    for j in neighbors(G, i)
        if not edgeDown(beta, i, j)
            optionScore = generalCase(G, beta, i, j)
            push!(options, (optionScore, j))
        end
    end
    sortBestFirst!(options)
    answer = 0.0
    probabilityToBeChosen = 1.0
    for (optionScore, j) in options
        answer += probabilityToBeChosen*optionScore
        if (i, j) in incidentVision(G, i)
            probabilityToBeChosen *= risk(beta, i, j)
        else
            probabilityToBeChosen = 0.0
        end
        if probabilityToBeChosen == 0.0
            break
        end
    end
    return answer
end
```

First, this function considers all the options for our step-after-next. It builds a list of these options, the "scores"–probability of eventual success–for each, then sums them together, following the analyzed incremental weighting. To that end, we must weight the score of the second best option by the probability that *both* the best option was unavailable *and* the second best option was available, and so on for the subsequent options. However, we must also remember to stop this summing process after the first "gamble," the best gamble, has been considered, since we will never *know* whether that edge is down, so we will never even attempt any edges worst than it. If it is up, good for us; otherwise, Lady Luck gets the better of us and we fail.

4.3 Complexity Analysis

On its own, each function we have covered here, save one, is computationally efficient.

The `baseCase` is a trivial "look up and return" function, as is the `generalCase` once it has updated the solution cache. Next, `expectedMaximum` and `incidentAverage` are both linear, each little more than "loop through a list once" and "calculate a weighted sum." Then there is `encodeInput`, also linear, which loops through a list of visitable edges and, not summing, concatenates a dictionary key, one charater at a time. However, we know from formal analysis that this algorithm-system is *necessarily* computationally expensive to execute. Several other helper functions are referenced in our example code here, such as `edgeDown` and `risk`, but these too are simple functions, simple enough to warrant exclusion here–their names describe their behaviors precisely enough.

All that remains that could contribute to the computational expense is `realizeFuture!`; it is, fittingly enough, the only function here with a `!` in its name.

Two things can lead to a hard to solve safest-with-sight problem: a large amount of nonincident vision at a single vertex; and a vertex that we can reach in two different paths with two different "accumulations" of knowledge. Our dangerous function contributes to the first of these plainly: the more nonincident vision, the more edges that will be `found`, and the more recursion that must be done. Since each step of the recursion calls the next step twice, we find the familiar $2 \times 2 \times \ldots$. Pity.

To understand how the latter, the "multiple paths/multiple accumulations," is produced by `realizeFuture!`, we must consider both how the value of `beta` stores which edges have been seen and how `encodeInput` controls the entries to our dictionary–a good measure of how much time we need is, easily, the size of the dictionary-like structure we must compute.

1. Path A through the graph is taken.
2. We are able to see set E_A of nonincident edges at the end of that path.
3. The value of `beta`, as `realizeFuture!` is called over time, as we are traversing that path, is changed.
4. The possible values of `beta` at the end of that path are contained in set β_A.
5. Each value of `beta` is given to `encodeInput`.
6. Each possible encoding produced by that function are contained in the set C_A.
7. Let's say the size of E_A is at most h.

8. The size of β_A is then at most 2^h.
9. The size of C_A is then at most the size of β_A, 2^h, since some encodings may be simplified together; for worst-case analysis's sake, we will assume the maximum.
10. Now a different path is taken, path B.
11. We see a different set of nonincident edges, E_B.
12. We have a different set of possible `beta` values, β_B.
13. We have a different set of possible encodings, C_B.
14. In the worst case, the 2^h encodings for A have no overlap with the 2^h encodings for B.
15. So, we have $2^h + 2^h = 2 \times 2^h$ encodings for these two paths.
16. We take another path, then another path, then another path, until n paths have been taken, each producing a different 2^h encodings.
17. In all, we have $n \times 2^h$ encodings, each requiring an answer to be computed before it can be added to the dictionary.
18. How big can h and n each be?
19. h can be at most the number of edges in the graph, say E.
20. And n can be at most the number of possible paths through the graph, equal to 2^{V-2}.
21. Therefore, 2^{E+V-2}, a very big number, entries may be needed.

All this complexity, measured in terms of how many different ways `encodeInput` might be called upon, stems from the different ways that `realizeFuture!` might behave, the different ways it can modify `beta` through the different paths we can take. It is the job of `encodeInput` to reduce the number of dictionary entries *as best it can*, but the future may still hold too many possibilities–a regrettable, unavoidable side effect of fate.

Chapter 5

Approximation

In computer science, we always consider the worst possible case. We must do so not because it is possible, but because doing any worst than that is *not* possible. We always know the *slowest* our problems will take us to solve. It is immensely regretful, though, that will oftentimes can never know the *fastest* we can do something. Perhaps this is for the best–it forces us to always *try* to do better.

There is an open problem that is famous in the discipline, namely P-versus-NP. Absolutely no one, in the history of mankind, has been able to give a rigorous proof of whether the problems that require at most n^k steps, for any fixed k, are fundamentally different from those that can be solved in n^k time if we had a machine that could explore unlimited potential solutions *simultanouesly.* What is more annoying, no one has even been able to prove that such a proof could even exist–we may never be allowed to know. Such are the dank confines of Plato's Cave.

The implications of this, for our primary discourse here, is that *there may be a faster way to solve safest-with-sight.* The limits of reason being as they are, though, we cannot say for certain. If we could make such a claim, either way, then P-versus-NP may be suddenly solved, an undertaking we are far underprepared for.

5.1 Our Limits

There are many ways to refer to the NP problems, such as "decision problems," "satisfiability problems," and "bin packing," but most importantly here, they are the *verifiable* problems. Imagine that the oracle imparts on us the solution to our problem, whatever that problem may be. We run back down from the mountain–where we trekked to listen to the oracle's wisdom–and happily tell our king what we have learned. He, however, is distrusting of the oracle, and asks us to double-check the oracle's answer. If the problem was so hard to solve in the first place that we even resorted to mysticism, we certainty can't verify the proposed solution by solving it ourselves and comparing numbers.

$$\text{Can you satisfy } E(X) = (X_1 \wedge \neg X_2) \vee (X_2 \wedge \neg X_1)?$$

For example, the "satisfiability problem," for which NP receives one of its names, goes as follows: given a boolean statement, is there an assignment for its variables such that the statement is true? In the worst case, we would need to imagine all true/false combinations of the n variables. Again, we go head-to-head with our villian. Yet, we can still verify the oracle's solution, although we are only given either a "yes" or "no" of whether the boolean statement is satisfiable–we just have to climb back up that mountain.

Say our variables are $X_1, X_2, \ldots$. If the oracle said the expression, say $E(X)$, was satisfiable, then we ask the oracle another question, "is $E(X)$ satisfiable if we set X_1 to true?" If she says "yes," then we know we can leave that variable fixed as true; otherwise, it must be false in all "satisfying configurations." Then, we repeat this process until we've found the correct value for each variable. Now with our "yes" and "no" responses transformed into a potential configuration, we plug those values into our original boolean statement and see whether the result is true. The number of steps required for this verification is relative the number of follow-up questions asked: $1 + 1 + \cdots = n$.

But if the original answer, the first answer our king asked us to double check, was "no," then the oracle's words simply must be trusted. Alas, the only way to prove a negative is through an exhaustion of possibilities.

Is safest-with-sight verifiable? If the oracle tells us, for free, as many times as we need, the value of x in expressions like $S(si, \xi) = x$, can we use this to verify that she tells the truth? It appears as though our problem is fundamentally *harder* than the NP problems–we are asking for more than a "yes" or a "no" response from the oracle, for a number that could be anywhere between zero and one. Safest-with-sight, therefore, is a *function problem*; how can we be certain though that it still can't be *reduced* to some weird, inside-out, unrelated decision problem like satisfiability?

Nonetheless, we can, in some cases, verify that the oracle is full of it. We do this by determining a lower and an upper bound that x must fall within in order to be correct. Of course, x cannot be lower than zero, nor can it be greater than one. However, if also cannot be lower than the probability of success of a *similar* safest-with-sight problem where we remove all lines-of-sight. More information can only *improve* the score of the ideal pathfinder, so it can obviously do no worse than if it never gained any information at all. Similarly, x cannot be greater than S would be if we give the pathfinder all the information there is to know up front so that it will only fail to reach the destination when no such path exists. Both of these values are easy to compute. We could even do better for the lower bound by removing the fewest lines-of-sight necessary to ensure a markovian system, the cases where our code runs quickly. In the appendices, I provide an extra test condition, showing that x can't be just *any* number within these bounds, but a number generated by a set of rules from β; the details, though, are merely technical.

So what if the oracle's x passes all of these checks? I don't know. These are not easy questions. So, for the meantime, we will keep ourselves to our humble sidewalk problem. We need accept that safest-with-sight may not be solvable any faster and proceed to approximate our solutions. On one hand, the villian has won; $2 \times 2 \times \ldots$ has been assured victory. On the other, who needs the sharpest needle in the haystack anyway? We can probably just find one that's good enough to sew with [18].

5.2 Ignorance is Bliss

It is common knowledge that, in the restaurant and web design industries for example, consumers suffer from the "choice paradox." When confronted with too many options, which, logically, should increase the quality of the choice they eventually make, the customer may instead feel overwhelmed and make poor, hastily made calculations. We appear to face a similar problem here.

When we are confronted with several lines-of-sight ahead of us, it can be difficult to resolve the number of possible realizations we may face upon reaching the vertex providing that vision. If we just forget about a few of those lines–slim down the menu or simplify the website–, then we can solve safest-with-sight quickly. It is remarkable, even, how only a few lines-of-sight take us out of a managable n to an intractable 2^n. Perhaps it is this sort of leap in difficulty that Oscar Wilde refered to when he said "brute logic"–by refusing to forgo a little information here and there, aren't we just making our lives needlessly hard?

Is the path we would follow, if we trust the solution to an "incident vision only" variant of whatever problem confronting us, really all that bad?

It is with this in mind that we propose these means of approximation for safest-with-sight:

- the "semirandom" solution, which always returns zero for edges known to be down and a random number–greater than zero and at most one–for all others;

- the “quick” solution, which removes all vision from a safest-with-sight problem before solving, thus minimizing the time needed to solve the approximation;
- the “incidence” solution, which removes all nonincident vision from the problem, which rectifies the issue of 2^h possible paths needing consideration;
- and the “hammy-down” solution, which removes all vision from subproblems starting from j that was not guaranteed to be passed along to it from any of its possible calling problems, those starting at i.

This last approximation scheme allows each subproblem to pass information along to its own subproblems, but only if that information is a “fitting hammy-down,” so to say. This ensures that each time we visit a subproblem for ij that we will do the same had we began our adventure at ij–in short, making the process markovian.

I expect that “quick” will be outperformed by “incidence,” but will that lose out to “hammy-down” in turn? It is hard to predict how the two will compare, since, although “hammy-down” retains more information, it also discards information throughout the computation process, something that is strange and alien to the pathfinders actual behavior. Is the benefit of more information worth it for the price of “impurity” we must pay to achieve it? Also, how would we even compare the approximations? If we select only one graph to run our tests on, then our comparison results will only be applicable for that one graph; but then if we run tests on an ensemble of randomly generated graphs, the process we use to generate those random graphs may impose a bias upon our results.

Before we can proceed then, these are pressing concerns that must be attended to presently.

5.3 Mr. President,

The simplest way, I have found, to work with random graph generation is to take an approach akin to that of Duncan Watts and Stephen Strogatz [20]. They defined a random graph process that produced "small world" networks. These are graphs that exhibit the anecdotal properties of "it's a small world after all," where the network is both highly clustered into distinct groups, or cliques, yet the distance between any two points tends to be small. This research ties heavily with the notion of "six degrees of separation," which posits that any two people on Earth tend to be *at most* six hops apart. One hop moves from me to my mother. Another, from her to her rabbi; then to his family friend in Pittsburg; to his neighbor, a professor at Carnegie-Mellon University; who had worked during her doctoral research years with the now Secretary of Defense; who, on occasion, has a meeting with the President of the United States.

Six degrees, each feasible, connecting me to President Obama.

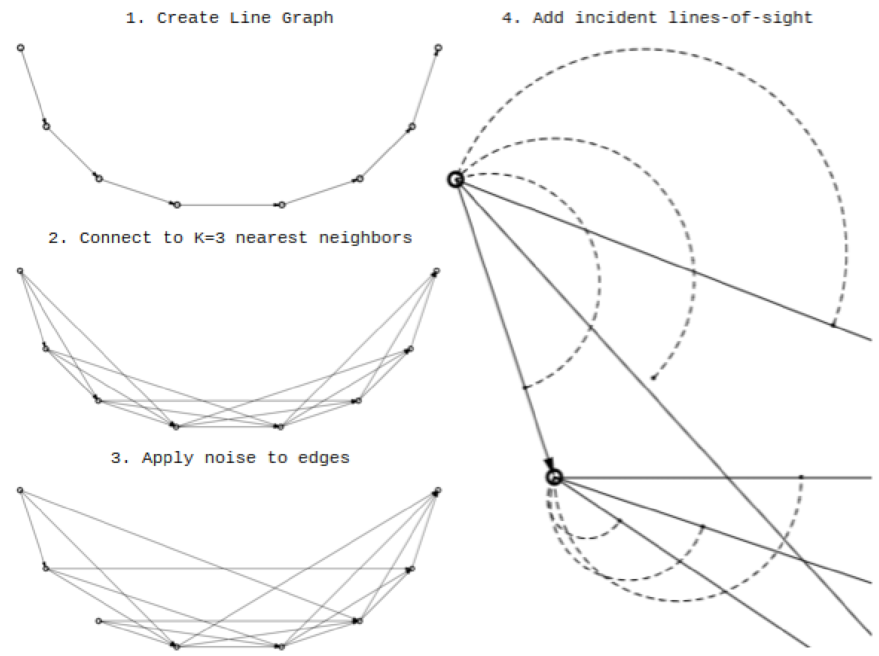
The Watts-Strogatz model begins by constructing a large graph with a regular pattern: the vertices are arranged in a ring and are connected to their K nearest neighbors, $K/2$ on either side. Next, with probability β–a different value than our β, only the letter, not the meaning, is in common–, each edge of the graph is randomly "rewired." So, if we are rewiring edge ij, then we choose a new edge ik that *could* be there, but *isn't yet*, and exchange ij for ik. When β is low, close to zero, the resulting graph is fairly regular; in contrast, when β is high, close to one, we see a more "random" graph. Somewhere in the middle, though, is a sweet spot for β–where "small world" networks come from.

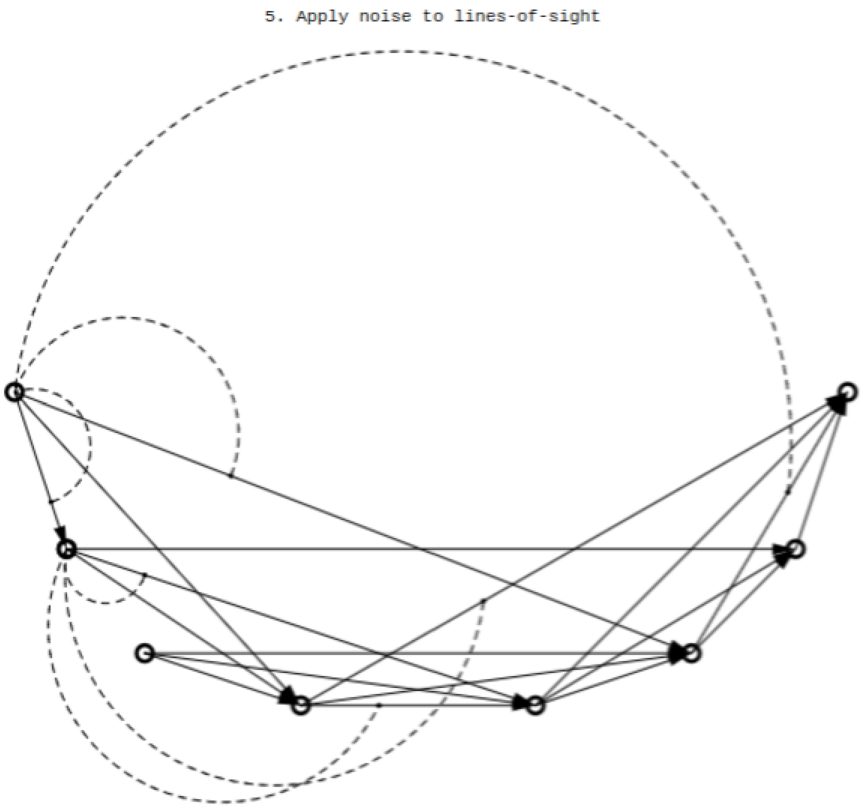
In a similar way, we can construct a random graph with vision.

1. Begin with $N = 100$ vertices laid out in a line.
2. Add an edge from each vertex to the $K = 3$ vertices after it, or all the vertices

after it when K is too large near the end of the line.

3. Rewire each edge with probabilility r_E.
4. Add a line-of-sight from each vertex to each of its incident edges.
5. Rewire each line-of-sight with r_W.
6. Set $\beta_{ij} = 0.25$ for all edges ij.


1. Create Line Graph
2. Connect to K=3 nearest neighbors
3. Apply noise to edges
4. Add incident lines-of-sight



This process should, I expect, produce graphs of varying "noise" in two different ways, "structure noise" and "vision noise." A fixed, but low value is chosen for the initial β values, since small chances of failure compound as the network grows large; similarly, K is chosen to be a small value to better reflect the properties of the real-world networks safest-with-sight will be primarily applicable for. You rarely come to an intersection with four options of where to go–there's left, there's right, and there's straight ahead.

What I plan to do with this process is to let r_E and r_W each be be 5, 25, 50, 75, or 95 percent, for all 25 different combinations thereof, and generate for each combination a hundred different random graphs. For each graph, we then follow the "instructions" given to us by each approximation scheme. We do this by asking the approximation scheme, what the probability of success would be through each of our possible next steps; given these values, we take (what the scheme believes to be) the best choice. After ending the trial for that approximation scheme, either in success or failure, we reset the trial, “forgetting” everything we’ve learned and moving back to the starting vertex, and continue with the next scheme. Trials would only be run on the approximations when at least one path exists to the destination; I am not setting my children up for failure–their success is nonetheless up to them still. My goal in all this is to determine which scheme tends to better guide our virtual less-than-ideal pathfinder to the destination. I hope to then be able to say, not, “this is better than that,” but, “in situations like this, choose that.”

5.4 Gearing Up

Coding “incidence” can be done in a few different ways. We could code an entirely new system based on the equations found for S earlier when incident-only vision is given. We could modify the graph that we’ve been given, then pass that as input to our existing code. Or, preferably, we will create a “drop-in replacement” for `realizeFuture!`, the function that has been causing us so much headache on its own.

```
function realizeFutureIncidence(G, beta, s)
    produce(1.0)
end
```

That’s it. Because this replacement just produces the likelihood 1.0, it doesn’t modify any β values; this already occurs, anyway, in incident-only vision cases, since `realizeFuture!` is only supposed to operate on nonincident edges. Therefore, all of the “heavy lifting” occurs withing `incidentAverage`, the bread and butter of “incidence.”

More work is required for the drop-in replacement for “hammy-down.”

```
function realizeFutureHammyDown!(G, beta, s)
    found = (-1, -1)
    where = "nowhere"
    for (i, j) in nonIncidentVision(G, s)
```

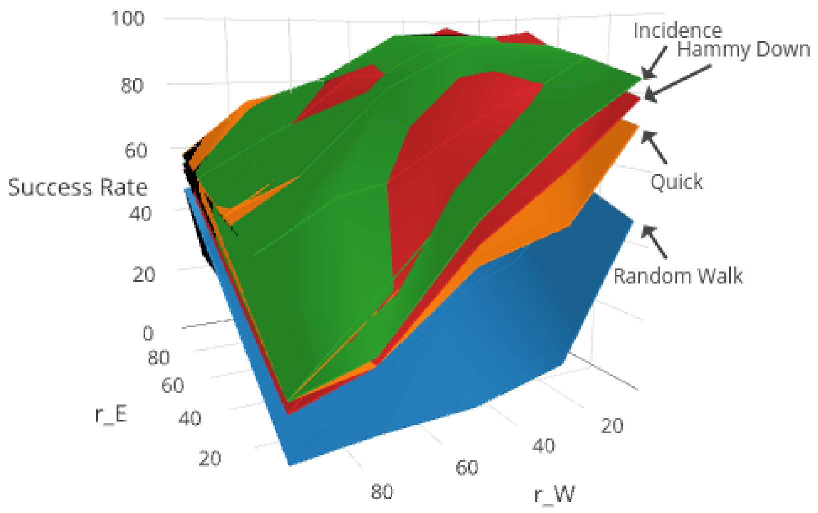
```
            if edgeUnknown(beta, i, j)
                if connected(G, beta, s, i, j)
                    found = (i, j)
                    where = "unknown"
                    break
                end
            end
        end
        for i in parent(G, s), (j, k) in nonIncidentVision(G, i)
            if edgeKnown(beta, j, k) and not visibleFrom(G, s, j, k)
                found = (j, k)
                where = "known"
                break
            end
        end
        if found == (-1, -1)
            produce(1.0)
        elseif where == "unknown"
            i, j = found
            temp = risk(beta, i, j)
            makeEdgeUp!(beta, i, j)
            for likelihood in @task realizeFutureHammyDown!(G, beta)
                produce(likelihood*(1-temp))
            end
            makeEdgeDown!(beta, i, j)
            for likelihood in @task realizeFutureHammyDown!(G, beta)
                produce(likelihood * temp)
            end
            makeEdgeUnknown!(beta, i, j, temp)
        else
            j, k = found
            temp = risk(beta, j, k)
            makeEdgeValue!(beta, j, k, 0.05)
            for likelihood in @task realizeFutureHammyDown!(G, beta)
                produce(likelihood)
            end
            makeEdgeValue!(beta, j, k, temp)
        end
    end
```

This function, `realizeFutureHammyDown!`, is not far from being identical to the original `realizeFuture!`. Much of the logic is the same, that of the `found`, but only when the edge that was found was in the "unknown" part of the graph visible by the vertex s. Otherwise, when the edge is found, but was seen earlier and is thus in the "known" part of the graph, yet is now "out of range from" s, we must handle things differently. It is at this latter step that the "destruction" and "impurity" of the algorithm occurs–we revert the β-value of the found edge back to 0.25, its original value when it was unseen, explore the graph further via a recursive call to `realizeFutureHammyDown!`, then return the found edge to whatever value it had when the function's execution began. In this way, we "forget" information whenever it goes out of our sight.

A watched pot never boils anyway.

5.5 And Ready to Go

The following is a *3d surface plot* of our results.



It is admittedly a mess on first glance. However, I feel that this method of display was the clearest to show what our experiment sought to discover: how the success rates of the approximation schemes, in general, compare. This data was inherently three dimensional, one for r_E, another for r_W, and a third for the variable of interest, the number of graphs out of a hundred for which each scheme guided us safely towards our destination.

To read this plot, begin at the annotations in the upper right of the image. following the arrows. We see that the “surfaces” corresponding to each scheme are neatly sorted in this corner of the plot, with “incidence” on top down to “semirandom,” the worst, on bottom. The higher the surface for a scheme is on the vertical axis, the more times it paid off to follow that scheme’s directions. As we move from the foreground of the image towards the back-left, we are moving from lower to higher values of r_E. The heights of the surfaces in the front, then, correspond to trials run on more orderly graphs and those in the rear to more random graphs. This increase in general success rates as we move away from orderly graphs may seem counter-intuitive at first, but consider how the more random a graph, the more opportunity we have for “shortcuts” here and there. Therefore, this increase in success can be attributed to a decrease in average path length. As we read from the right to the left, r_W increases, so the heights on the right correspond to graphs closer to incident-only vision, and those of the left to graphs with almost no incident vision at all. The results here are as expected–the less incident-vision, the more gambles we must make, and the less certainty we have in reaching our destination.

Although there are regions of the surface plot where “hammy” shows through on top, suggesting that those are regions where this scheme is the best, we could attribute these winnings to chance alone. It is difficult to say, with this data, whether “hammy” really is the best choice in the $r_W \approx 0.50$ region, which intuitively *could* make sense–“hammy” does, after all, try to strike a balance between incident and non-incident vision while maintaining Markovianness. But, for *so much more* of the plot, green comes out on top, suggesting “incidence” really is the best of the approximation schemes proposed herein. The occasional points where red, or even orange, in the back left corner, show through on top, could all be little more than “noise,” a regrettable side effect of stochastic modeling.

So who cares if “hammy” *might* be slightly better around $r_W = 0.50$? It takes 2^h longer to solve, so why not just use “incidence” and be done with it?

More importantly though, what does it mean that “incidence” is usually no worse than “hammy?” For this question, we need to consider what it means for “hammy” to be a good approximation for the exact solution: *any naturally markovian safest-with-sight system is exactly solved by “hammy.”* Since markovian is defined by having all routes to each point producing the same vision set, “hammy” will never remove vision when it recurses into subproblems, since any vision j will have it will have regardless of the i that led into it. So, “hammy” never modifies the information in any way and must behave, precisely, like the naive, exact solution–the one that is typically hard to calculate.

Therefore, since “incidence” is typically just as successful as “hammy,” “incidence” must tend to be a good approximation in *any* markovian or near-markovian setup.

Finally, the non-markovian cases, corresponding to the surface heights along the left edge of the plot, prompt us to ask: are these *naturally harder to be successful on*, meaning that the success rates for the exact solution would also “dip” strongly there–taking on a similar shape as the approximations–, or are they just *naturally harder to solve*, meaning that the exact-surface would not drop down on the left–taking on a distinct shape of its own, tending to be higher, flatter, further out of reach, beyond the trappings of complexity set out by the antagonist, $2 \times 2 \times \ldots$?

Chapter 6

Survey

Before I conclude, I feel the need to say, “we are not alone,” if only for one last mention of alien abductions. Uncertainty, prediction, probability, statistics, randomness, oracles, decisions, path-finding, expected maximization, and, all of it really, is nothing new to Man. These topics have interested us since Antiquity–the term “oracle,” a formal term used in computer science proofs stems, perhaps, from the mythological use of the same idea, of a being who gives boundless knowledge, oftentimes at a price.

What I have collected in these sections are those resources which have aided this research the most. Although the final approach was much our own, pieces of it–either bits of notation or just ways of looking at the problem–would never exist had I not stumbled upon these papers.

Presently then, let’s give these works what they are due.

6.1 Leeland on Case Based Reasoning

At one point, primarily to extend the scope of this work for the sake of conference submission, I entertainted the thought of designing a *cased-based reasoning* system for approximations. Leeland [12] collects nine papers by various authors on this topic.

Case-based reasoning, often abbreviated CBR, is a technique for approximating or bootstrapping solutions to problems of high complexity or dimensionality. This is done by comparing subproblems to similar subproblems that have been already solved, or cases. A common theme among the nine papers is the four Rs of case-base reasoning, which act as a general framework for all its applications:

- Retrieving the most similar case to the given problem from memory;
- Reusing the information about the case's configuration and solution for solving the given problem;
- Revising that solution to better fit the given problem; and
- Retaining new information gained through this process about the given problem in memory.

In general, individual papers from this collection seek to either (i) show careful application of this framework to a particular problem, determining methods of quantifying distance/similarity between cases, regression methods used to approximate a new solution from old solutions, and storage/retrieval schemas; (ii) further formalize the four Rs, such as one article accomplishes by relating the process to *absorbing Markov chains*; or (iii) survey the history and trends of the topic.

Further common themes in the collection include:

- comparison of reasoning models;
- case encoding/decoding schemes;
- regression techniques;
- probabilitistic and fuzzy techniques; and
- parameter-tuning techniques.

Regretfully though, time constraints prohibited further application of this paradigm to safest-with-sight.

6.2 Beneroya on Convolution Integrals

Once I had resigned us to our fate of coming at this problem with a solely analytical attack, I realized that we needed to be equipped with the right weapons–the formalities of probability theory. Beneroya [2] provides a thorough, yet general, introductory text for probability modeling in engineering and science. The first four chapters lay the groundwork for set theory, probability theory, and random variables, modeled as probability density functions. This follows how most texts commence the subject.

The remaining chapters explore systems and processes of random variables and their dynamics, paying close attention to their formalizations, analyses, and applications. For example, vibrating systems, those that can be modeled using a single degree of freedom to describe motion, can be analyzed using convolution integrals.

$$\int_{-\infty}^{\infty} F(\tau)G(t-\tau),\, d\tau$$

Such integrals are used to combine multiple random variables, resulting in a description of the system's overall behavior. Analyzing systems of more than one degree of freedom can be done with similar convictions by writing the system in a matrix/vector form, naturally leading into discussions of *eigenvalues* of random distribution.

Systems with dynamics that are continuous in both time and one-dimensional space can often be modeled as or with analogy to taut strings and vibrating beams, either axially or transversely, particularly with how such systems oscillate and respond to forces. Reliability of a system can be modeled through the analysis of when or how often a system will enter undesirable states, such as a stress or capacity level falling too high or too low. Nonlinear dynamics, which should be familiar to any reader already familiar with *chaos*, are grounded by Beneroya also in terms of density functions. Nonstationary models, those in which the distributions of random input variable may vary with time or space, are of particular interest anddifficulty due to the amount of data needed, but may be approximated by building up stationary models, like bricks to a building.

Beneroya's discussion next leads into *Monte Carlo* methods, in particular how this method can be used to solve complex models. I must note that the text did not mention *Monte Carlo Markov Chains* as they are used in Bayesian approaches, but the connections between the two will be clear for any statistical-savvy reader. And, finally, the author shows that fluid dynamics under perturbations can be modeled as a system that is continous both in time and multi-dimensional space.

Much of this text is based on modeling real-world problems as density functions and on techniques for combining multiple input distributions together to result in a single output distribution. This, one hopes, perfectly describes the behavior of the system of interest, much in the same vein as the analyses carried out in a previous work of mine [11] on the antifragilty of hypothetical rabbit populations.

Before proceeding, I bring attention to a practice carried out in this text that I have not witnessed elsewhere: whenever a proper noun is first used in the name for a term (e.g., Marvok chains), a substantial section is included nearby that features, when available, a photograph of the namesake (e.g., Markov himself) always along with a detailed history of that person's life and contributions to math and science. I enjoyed these asides and would like to thank the authors and editors of this text for their inclusion.

6.3 Dreyfus on Dynamic Programming

The closest works to ours, those of Dreyfus [7] and Bertsekas [3], are two texts in the Mathematics in Science and Engineering series. Although old–respectively, they are from 1977 and 1976–, they have proved particularly relevant to our research, together covering the full scope of mathematics required to treat safest-with-sight.

Dreyfus grounds dynamic programming in terms of pathfinding, using a notation throughout for selecting the minimum of two values, x and y, written compactly as the following.

$$\min \begin{bmatrix} x \\ y \end{bmatrix}$$

This is particularly useful for fitting on the page the minimization of two long equations, such as the following, an example also using a generic function S, representing the solution to the problem under some particular input (x, y).

$$S(x,y) = \min \begin{bmatrix} a+b+c+d+e \\ f+g+h+i+j \end{bmatrix}$$

For cases where minimization must occur over multiple terms, the following sort of notation is used instead.

$$S(x,y) = \min_{k=1,\,2,\,\ldots,\,N} [a_k]$$

In this way, Dreyfus builds up ever-simple examples of non-random problems that can be stated in terms of optimal pathfinding on directed graphs, either cyclic or acyclic. Once this groundwork has been laid, stochastic modeling is introduced, requiring only simple (rational) probabilities from which weighted sums can easily be expressed without requiring densities or measures. Multiple *control policies* are explored and compared: is it better to minimize the maximum error or to maximize the expected payoff? Different problems call for different controls. Following that, special cases are discussed where particularly useful techniques can be employed.

In problems where the state can be said to move to one of a set of subsequent states, each with an associated probabilistically, regardless of previous states explored, we can model the system as a transition probability matrix and use Markovian techniques. This requires determining the probabilities associated with such transitions, but allows for fast, linear, matrix-based algorithms.

Consider problems where we can model the solution as follows, where $V_i(x)$ represents some optimized expected value of stage i in state x, $B(i)$ is the set of random variables associated with i, and f is some function quadratic with respect to the elements of $B(i)$. Where boundary cases $V_N(x)$ are defined, Dreyfus analyzes such systems by induction.

$$V_i(x) = \min_y \left[\sum_{B(i)} f(B(i)) \right]$$

Next, consider systems where the decision maker is given information after each decision made, effectively allowing it to "change its mind" as it goes. This is notably similar to our problem, and can be refered to as adaptive control, dual control, or bayesian control, all based on different ways of framing the problem, but all ultimately the same thing. Here, the author uses a technique we have become familiar with on our own right: recasting condition probabilities such as $P(a \mid b_1, b_2, \ldots, b_n)$–that is, where the probability of a single event given multiple other events–explicitly as in the following.

$$\begin{aligned} P(a \mid b_1, b_2, \ldots, b_n) &= \frac{P(a, b_1, b_2, \ldots, b_n)}{\sum_{a'} P(a', b_1, b_2, \ldots, b_n)} \\ &= \frac{P(a, b_1, b_2, \ldots, b_n)}{\int_{-\infty}^{\infty} P(a', b_1, b_2, \ldots, b_n)\, da'} \end{aligned}$$

Normally in probability treatments, the denominator is left simply as $P(b_1, b_2, \ldots, b_n)$ in either case, but the explicit noting of an iteration may better correspond with the computations required. Further connections could be made to

Beneroya's convolution integrals to demonstrate the benefits of Dreyfus's notation. As used in the current discussion, probability densities are computed in this way to solve for expected values given some decision is assumed to be taken by the decision maker; that which optimizes the expected value is taken as the solution.

Note that this precisely describes the general approach we have taken in solving safest-with-sight.

6.4 Bertsekas on Stochastic Modeling

Bertsekas [3], in contrast to Dreyfus, approaches the stochastic dynamic programming class of problems not from pathfinding up, but from measure theory up. These differences allow Dreyfus to arrive at implementations of solutions quicker, but Bersekas, and ourselves, to arrive at axiomatically-grounded proofs quicker.

This work begins with a clear introduction formalizing optimality in terms of total orders, where information is freely available, and partial orders, where information is imperfect. Random variables, responses from other "players," imperfect information, and so on are all described as events "chosen" by Nature, a well-known classic interpretation in game-theoretic models. By considering expected values of utility functions over all possible outcomes in the outcome set, we can develop a probabilistic total ordering (assuming some notion of tiebreaking is given), where $i < j$ whenever $E\left[U(i)\right]$ $< E\left[U(j)\right]$. That is, we arrange our choices in order of their average values.

Still within the introduction, sequential decision models, stated as the primary focus of the remainder of the text and which can represent the adaptive control scenarios treated in Dreyfus, are modeled by an iterative process:

- $x_{k+1} = f_k(x_k, u_k, w_k)$, where
- x_k is the state of the system at discrete time k,
- u_k is the decision made by the decision maker according to some component

function $u_k = \mu_k(x_k)$ at time k, and

- w_k is Nature's input to the system at time k.

Note that the transition functions, as well as some underlying utility function, may vary at each time step. An overall utility function can be given, then, as a summation of all component functions over time: $U(\mu, u, w) = \sum U_k(x_k, \mu_k(x_k), w_k)$. Finally, the general sequential decision making problem reduces to finding some set of functions μ that optimizes the expected value of U–expected value because w will be "chosen" by Nature or Lady Luck. Additionally, akin to the form used by Dreyfus, this work prefers notation such as the following.

$$J(x, y) = \inf_{k=1,\,2,\,\ldots,\,N} [a_k] \tag{6.1}$$

Infima are prefered to minima except in cases where existance of lower bounds either are obvious or have been explicitly shown, a formal technicality.

In following chapters, the work proceeds by laying out the conventional set theoretic basis for measure theory and its application to probability theory. Critical note is made that modeling in this way requires countability to either be shown or assumed; for some applications of measure theoretic probability, this can be almost negligibly trivial, and in other cases (involving infinities or continuous intervals, for example), substantial work may have to be done to validate proofs. In our work the former holds, the countable, discrete nature of graphs working to our advantage, so I, as Bertsekas does, let countability be assumed except where noted.

The general technique above is applied out under different perfect information classes of problems where different assumptions hold, leading up to considerations about the imperfect information classes. Reduction, in these cases, is performed to a weighted sum of similar perfect information cases. Then, in manners again much like

Dreyfus, albeit breifer, this work explores solving linear systems with quadratic controls and finite state markov chain models. Several other examples are given, showing analyses involving expected values over finite spaces that in turn are approached by linear algebraic techniques.

An issue rising out of the reduction to perfect information cases is that the iteration over all possible values can suffer from the curse of dimensionality. Approximations, then, must be performed in order for the result to be computable. We encountered this problem ourselves–at each step we must consider the 2^h "what ifs" that might occur.

This work next explores discretization of continuous intervals, where convergence towards the continuous solution is often the primary topic of analysis. Of more relevant interest to safest-with-sight, a definition of *adaptive* controllers is given, being informally those that are not optimal but still use the "measurements" collected "as it goes" to some advantage.

Two fundamental controllers are given to illustrate this class of controllers: the naive feedback controller, which simply assumes at each step that Nature inputs an expected value of some distribution; and the open loop feedback controller, which calculates probable inputs from Nature given past inputs, but assumes that no future information will be collected. Neither of these approaches to control have proved useful for our application, since neither makes sense for "all or nothing" failures– there is no "average" between up and down, and the probabilities between failures are all disjoint, so the latter type of controller has nothing to learn based on Nature's past.

The final chapters of the text revisit several of the classes of problems already discussed here, above in this section and previous, but in the countably infinite scenarios. Much of the distinction between this section and what has already been recounted is technical and to summarize it would be, for the most part, repetition.

"Well tell me about Bertsekas's appendices," you say. "Well of course," I say.

The first sets out mathematical foundations, in the conventional sense, for set theory, countability, spaces, topologies, matrices, compact sets, continuous functions, convexity, and so on. The second, of more interest to safest-with-sight, sets out theorems for existance and necessary and sufficient conditions for optimality:

- any finite set has a minimum,
- any infinite set X has a minimum if there exists a continuous $f : R^n \to R$ and X is a compact subset of R^n, and
- when $X = R^n$, $\nabla f(x^*) = 0$ is required to show x^* minimizes $f(x)$.

The third appendix lays the groundwork for measure theoretic probability theory. A probability space is given by (Ω, F, P), where:

- Ω is the set of possible outcomes, F is the set of possible events, and P is the probability measure;
- F is a set of subsets of Ω that contains Ω, $\bar{A}$ if it also contains A, and $\bigcup_k A_k$ if it also contains each of the countable and disjoint sets A_k; and
- P is a measure mapping F to the interval $[0, 1]$ such that $P(\omega) = 1$, $P(\emptyset) = 0$, and $P(\bigcup_k A_k) = \sum_k P(A_k)$ for countable and disjoint sets A_k.

A random variable, unlike in other works that first treat them using density functions, is a function that maps Ω, the outcomes, to R, the real numbers. In this sense, the random variable associated with a die maps the physical orientation of the die after being tossed to a numeric value. A *cumulative* density F can then be defined for a random variable x in measure theoretic terms: $F(z) = P(\{\omega \in \Omega \mid x(\omega) \leq z\})$.

The final appendix formalizes finite state markov chains. In a stationary case, we are given a single $n \times n$ matrix p where, for each element, $0 \leq p_{ij} \leq 1$ and each row and column sums to some value π also in that interval. Then, the probability that the system will enter state s_j next is given by $P(x_{k+1} = s_j \mid x_k = s_i)$, equal simply to p_{ij}.

In the nonstationary case, we are given a similar matrix p_k for time step k. The probability that the state will transition from i initially to j at time k can be given by $(\prod_{m=1}^{k} p_m)_{ij}$. This reduction is just as in the stationary case, thus showing the author's reasoning for using only stationary finite markov chains elsewhere in the text.

A markov chain p is *ergodic* if it is possible to go from every state to, via some path, to every other state; we also call these irreducible chains. The first passage time of s_j is defined as the time step k when the system first enters state s_j. The mean first passage time is the expected value of k and is, a random variable in its own right.

The final proof of the text, within this appendix, is one relating to an assumption made in an earlier chapter about there being a state s_* such that the mean first passage time of s_* is not infinite, implying that s_* is *almost certainly* accessible by the system, albeit not necessary likely.

6.5 Aboudolas on Rolling Horizon

More recently, Aboudolas published a paper applying rolling horizon to the management of traffic congestion [1]. Solving the traffic control problem is nontrivial, since balancing the relationships and expectations between each redlight is a particularly difficult problem. This paper begins by discussing the various strategies that the literature has proposed and studied, sometimes even tested in the real world by city officials. The means of these strategies are wide in comparison, but the general scheme is: each light has sensors with which to collect, store, and share data with traffic control. How traffic control responds is what is meant here by "strategy."

Aboudolas adopts rolling horizon decision making for this task. A *model-predictive* scheme is proposed that behaves as follows:

1. At time step k_0
2. we solve a quadratic programming control problem
3. based on a measured or estimated input, $x(k_0)$,
4. and also based on available predictions of traffic demans, $d(k_0)$.
5. A control "trajectory," or sequence of proposed actions, is calculated,
6. based on the solution to that problem and estimates of x and d at future time steps.
7. However, we only put into action the first few steps–the rolling horizon–of that trajectory.

Although the problem and perspectives taken by Aboudolas do not match those of our discourse here, on safest-with-sight, this prediction-based movement matches our general strategy exactly: the entire future is planned for, but only a single step is taken at a time before we must respond to the new information we collect, resulting in the preference of a policy over a fixed path.

6.6 Carey on Network Performance

Much work exists on the subject of networks, what happens when they fail, and how we can prevent those failures. Carey gave a thorough paper on the subject [4]. Fortunately, this paper is also closely relevant to the probabilitistic perspective taken for solving safest-with-sight.

For Carey, a network had N vertices and a set of directed edges connecting those points, denoting A. These edges represented branches in waterways, and the event that a branch would become obstructed due to a seismic event was given by the variable θ_a.

$$\theta_a = \begin{cases} 1 & \text{Link } a \text{ is up} \\ 0 & \text{Otherwise} \end{cases}$$

The probability of each edge's being up is seen as the expected value of this variable for that edge over all possible realizations. This work then concerned itself with the flow patterns that the water would take through the network. The maximum flow, for a given realization of θ, is denoted $MF(\theta)$, and the expected maximum flow, a known computationally hard problem [19, 17], would be denoted $\mathbf{E}_\theta MF(\theta)$.

Although calculating this value exactly is intractable, Carey attempts to find closer bounds on that value than previous works. Ideally, this range will be narrow, permitting a decision maker to select its midpoint as an approximation to $\mathbf{E}_\theta MF(\theta)$.

Parallel considerations are given, as the paper continues, to optimal flows when paths must be fixed in advance and the economic benefits that arise from the relationships between parties. The now-calculatable bounds on these values are then applied to a toy model based on Salt Lake City. Still, though, this work does not contain a component analogous to our lines-of-sight.

6.7 Nickerson on Decision Modeling

Nickerson says in his paper, "Prior to receiving the information contained in Y we must weight the expected value of each alternative by the probability of receiving Y" [14]. Written in 1980, in the same operations-research-laden era as Bertsekas and Dreyfus, Nickerson was setting forth a way of modeling decision modeling itself. In this, the context of Y is given as *imperfect information*–neither do we know nothing, nor do we know everything, but only what the oracle, Nature, and Lady Luck would have us know. Its use here is in calculating the "expected value of modeling," a problem of considering, as a decision maker, which models to use in our problem solving process and how to update the information our models previously gave.

In all, this is perhaps the most abstract paper I have encountered during this research, but in a way it captures all of safest-with-sight in its thesis, that “the value of the model to the decision maker is in the information provided about the problem.” The analogy between our two works is more apparent when I say, “the value of the next step to the ideal pathfinder is in the information provided about the paths ahead.”

Nickerson’s work strives to rationalize, rigorously, the close interaction between the decision maker–the boss in charge–and the analyst–the recently-hired number cruncher. There, the two are caught in a ballet, one making decisions, leading to new information being gathered from the visible world, and the other, fitting those values into her analyses, responding to the boss with these predictions about the “invisible world.” In pathfinding, these are naturally joined, the pathfinder serving both roles. It steps, observes, and considers, repeating as necessary.

6.8 Almost There

For additional survey, I refer the reader to the preliminary works, [10, 11]. Because (i) the texts from the 70s, [7, 3], cover all the necessary techniques and (ii) I have been unable to find works more specific to our formulation of the stochastic pathfinding problem than those treated above, I do not recreate those surveys here. This should provide a more unified, concise reference for the reader.

Chapter 7

Conclusion

We have arrived at our destination. The path we have taken to get here has been fraught with measure theory and other dangers. But we kept our smarts about ourselves, and we found a solution to our problem–we solved $S(ij, \xi) = x$. We did this in the simplest of cases, when our starting point and our destination were in close proximity, we did this when our path ahead is straight, the only difficulty being commitment; and we did this for the general case, for all cases.

But we can never, I fear, actually compute that solution. Instead, we could only have feasibly gotten here, this, the destination, the other side of campus, through the swamps of summer construction projects, by approximation. We produced a 3d surface plot. This graphic suggested that the approximation scheme, named “incidence,” which cares only about vision if it is of an incident edge to the current position, was the best of those proposed here, beating out three other contenders, “semirandom,” “quick,” and “hammy.”

Yet more can be gleaned from the surface plot. The stories it has to tell aren't over.

Turn back to the plot and look closely, notice that all of the surfaces behave similarly. Although their heights differ consistently throughout, the curves they exhibit across the 3d space are nearly equivalent. On the side of high structure and long distances, success rates are low, but as the structure of the network is scattered by the increase in r_E, this success rate likewise increases. I expected as much.

Along the other dimension, of r_W, the curves are more interesting. On the side of hight incident vision–when most information is immediately available for "consumption"–, the success rates are significantly higher, regardless of approximation scheme. Even "semirandom," which always takes a random step *except those known to be down*, sees a marked performance increase. And, to no surprise, "incidence," if for no reason other than its namesake, performs clearly better than the other schemes as information is mostly incident.

But the other two schemes, "hammy" and "quick," also perform well here, yet do not have any direct mathematics regarding incidence or non-incidence. The latter doesn't even take vision into account at all! To understand this, think, "how exactly were these schemes used in the simulation that produced this plot?"

Although "quick" operates by proposing the best fixed path towards the destination, it still only ensures that the first step of that path will actually be traveled. Once the simulated pathfinder has taken that step, it is given more information about the problem–the values of β used by "quick" are updated–and the scheme is again invoked. Because of this, the path proposed by this approximation scheme at the first step may not be the path traversed by the pathfinder in the end.

How, then, does the amount of incidence of the graph tie in with the natural dynamics of the "step, observe, plan" pattern? It would appear that having information about a future decision is less conductive of success simply due to the fact that that future decision is less likely to be taken. Therefore, can I conclude that decision making, when a goal is in mind, coupled with the ability to choose *how* we observe, should be done in a way that observations are "immediately consummable," observations whose values play a role in the choices we will soon be considering?

We, of course, cannot make such claims certainly yet. More work will have to be done to solve our open problems:

- What shape does the exact solution produce on the surface plot?
- Is safest-with-sight NP? NP-Complete? NP-HARD? #P?
- What is it about incident vision that makes pathfinding so much easier?

What I will say, though, is what we have, throughout this discourse, touched upon about decision making in general. Simon described the thought-process as pathfinding through an abstract space [18]. He posited that our actions and facilities are simple machines; the only reason we are apparently complex actors is because we are acting in response to a complex world. So we too wish to adopt this philosophy–decision making as pathfinding. The world we behave in has an inextricable hold on our behaviors, just as the structure of a graph and its lines-of-sight have on our ability to successfully traverse it. In our limited capacity, as humans, as simple machines, to understand the "graph of the world," so to say, what is it we should pay attention to? What features of decision making, of pathfinding, are most conductive of success, of a good life?

Don't gamble, if you can avoid it. Don't cross edges that you haven't seen yet, whose missteps that will cut our traversals short.

Don't overcommit, if you can avoid it. Even if you stick to your observations, starting down a route that you've only partially observed and that provides no escape is no better than a gamble. Its adversity is only *delayed* until a downed edge later becomes inevitable.

And don't ever tempt the oracle. Information about the far future is only useful when that future is certain–in which case, who cares if we know what that future has to hold. Either we have a say in the cards we're dealt, or we are along for the ride. Regardless, the Unknown is ahead of us. It is unimaginably large, holding untold secrets, ills, and fortunes. And it begs to be explored.

Bibliography

[1] K. Aboudolas, M. Papageorgiou, A. Kouvela, and E. Kosmatopoulas. A rolling-horizon quadratic-programming approach to the signal control problem in large-scale congested urban road networks. *Transportation Research C*, 2009.

[2] Haym Benaroya, Seon Mi Han, Seon Mi Han, and Mark Nagurka. *Probability Models in Engineering and Science (Mechanical Engineering (CRC Press Hardcover))*. CRC Press, 2005.

[3] Dimitri P. Bertsekas. *Dynamic programming and stochastic control, Volume 125 (Mathematics in Science and Engineering)*. Academic Press, 1976.

[4] Malachy Carey and Chris Hendrickson. Bounds on expected performance of networks. *Networks*, 14, 1984.

[5] Bill Cherowitzo. Richard bellman's biography. *Salvador Sanabria*, 2003.

[6] A.D. Dimitriadis, N. Shah, and C.C. Pantelides. Rtn-based rolling horizon algorithms for medium term scheduling of multipurpose plants. *Computers chem. Engng*, 21, 1997.

[7] Stuart E. Dreyfus and Averill M. Law. *The art and theory of dynamic programming, Volume 130 (Mathematics in Science and Engineering)*. Academic Press, 1977.

[8] Malcolm Gladwell. The Gift of Doubt. *New Yorker*, June 2013.

[9] Gregory A. Godfrey and Warren B. Powell. An adaptive dynamic programming algorithm for dynamic fleet management, i: Single period travel times. *Transportation Science*, 36, 2002.

[10] B. Knowles and M. Atici. Fault-Tolerant, but Paradoxical Path-Finding in Physical and Conceptual Systems. *ArXiv e-prints*, June 2014.

[11] B. A. Knowles. Initial Analysis of a Simple Numerical Model that Exhibits Antifragile Behavior. *ArXiv e-prints*, August 2015.

[12] Antonia M. Leeland. *Case-Based Reasoning: Processes, Suitability and Applications (Engineering Tools, Techniques and Tables).* Nova Science Pub Inc, 2010.

[13] Alexis Madrigal. Meet the Robotics Company Apple just Anointed. *Atlantic*, June 2013.

[14] Robert C. Nickerson and Dead W. Boyd. The use and value of models in decision making. *Operations Research*, 28, 1980.

[15] Mehrdad Niknami, Samitha Samaranayake, and Alexandre M. Bayen. Tractable pathfinding for the stochastic on-time arrival problem. *CoRR*, abs/1408.4490, 2014.

[16] Suresh Sethi and Gerhard Sorger. A theory of rolling horizon decision making. *Annals of Operation Research*, 29, 1991.

[17] Yuhong Sheng and Jinwu Gao. Chance distribution of the maximum flow of uncertain random network. *Journal of Uncertainty Analysis and Applications*, 2, 2014.

[18] Herbert A. Simon. *The Sciences of the Artificial - 3rd Edition.* The MIT Press, 1996.

[19] Steven Stern. Approximate solutions to stochastic dynamic programs. *Econometric Theory*, 13, 1997.

[20] Duncan J. Watts. *Six Degrees: The Science of a Connected Age*. W. W. Norton & Company, 2003.

Appendix A

Proofs

Definition 1 (SWS Problem)**.** *An SWS problem is defined by (i) an acyclic digraph $G = (V, E, W)$, (ii) an edge ij, a destination vertex d, (iii) a collection of probabilities of failure β, (iv) a set of mutually exclusive events ξ, and a probability measure P. Where G, β, P, and d are understood, an SWS problem is denoted $S(ij, \xi)$.*

If $a \in V$, then there exists a vertex a in the graph. If $ab \in E$, then there exists a directed edge from a to b and $a < b$. If $abc \in W$, then there exists a line-of-sight from a to bc and $a \leq b < c$.

The events of P are π and for all $ab \in E$, α_{ab} and δ_{ab}, as well as their compliments, denoted with an overbar. These respectively represent the event that a pathfinder could traverse the graph from i, immediately to j, then through some path to d without encountering a deadend or cross a failed edge; the event that edge ab has failed; and the event that the pathfinder included ab in its traversal. Given β, $P(\alpha_{ab}) = \beta_{ab}$.

When the pathfinder reaches or begins its traversal at vertex a, it is informed of the failure status of each edge bc where $abc \in W$. The pathfinder is assumed to traverse edges optimally for what it knows about the graph via G, β, and the information given it through W.

The solution to an SWS problem is the value of the call to S in the expression

$S(ij, \xi) = P(\pi \mid \xi, \delta_{ij})$.

Definition 2 (Expected Value)**.** *In the context of $S(ij, \xi)$, the expected value operator $\mathbf{E}_{\xi'}$ is defined by $\mathbf{E}_{\xi'} f(\xi') = \sum_{\xi' \in R(\xi)} P(\xi' \mid \xi) f(\xi')$, where $R(\xi) = \{\xi' \mid a_{bc} \in \xi' \text{ iff } (a_{bc} \in \xi \text{ or } jbc \in W)\}$ and a_{bc} is either α_{bc} or its compliment.*

Theorem 1 (Inductive SWS Solution Theorem)**.** *If $j = d$, then $S(ij, \xi) = P(\bar{\alpha}_{ij} \mid \xi)$. Otherwise, $S(ij, \xi) = P(\bar{\alpha}_{ij} \mid \xi) \mathbf{E}_{\xi'} \max_{jk} S(jk, \xi')$.*

Proof. The base case is trivial given the definition of the problem. Since $P(\pi \mid \xi, \delta_{ij})$ assumes ij is being crossed, we must conclude that all edges previously crossed were done so without failure; otherwise, ij would not have been available for selection. The only way the pathfinder can fail now is if this last edge itself fails; therefore, its probability of success is just the probability of that edge not failing.

For the general case, we assume that all jk subproblems have been calculated properly. We also must assume that all edges previously crossed did not fail in the same manner as before. The pathfinder still must cross edge ij without failing and continue. These are disjoint events, so the "and" here is a simple multiplication. The probability that it crosses the edge without failure is analogous to the base case, and the probability of continual success is that of the next edge, jk, that maximizes S here. This follows from the definition of the pathfinder's behavior. Therefore, we find this by an expected maximum operation.

$\square$

Definition 3 (Nonincident Expected Value)**.** *In the context of $S(ij, \xi)$, the nonincident expected value operator $\mathbf{E}_{\tilde{\xi}'}$ is defined by $\mathbf{E}_{\tilde{\xi}'} f(\tilde{\xi}') = \sum_{\tilde{\xi}' \in \tilde{R}(\xi)} P(\tilde{\xi}' \mid \xi) f(\tilde{\xi}')$, where $\tilde{R}(\xi) = \left\{\tilde{\xi}' \mid a_{bc} \in \tilde{\xi}' \text{ iff } (a_{bc} \in \xi \text{ or } [jbc \in W \text{ and } j \neq b])\right\}$ and a_{bc} is either α_{bc} or its compliment.*

Definition 4 (Ordered Neighbors)**.** *In the context of $S(ij, \xi)$, iif $jk, jm \in E$ and $P(\pi \mid \xi, \delta_{ij}, \delta_{jk}) \geq P(\pi \mid \xi, \delta_{ij}, \delta_{jm})$, we say $jk = r(a)$ and $jm = r(b)$, where $a < b$*

and the domain of r *is the integers* $1, 2, \ldots, deg(j)$.

Definition 5 (Best Gamble)**.** *In the context of* $S(ij, \xi)$*, iff* $\xi \cap \{\alpha_{jk}, \bar{\alpha}_{jk}\} = \emptyset$*,* $jjk \notin W$*, and, for all* jm *where* $jm = r(a)$ *and* $a < g$*, either* $\xi \cap \{\alpha_{jm}, \bar{\alpha}_{jm}\} \neq \emptyset$ *or* $jjm \in W$*, we say* $jk = r(g)$ *and reserve the name* g *for this usage.*

Lemma 1 (Incidence and Subproblems)**.** *If* $r(1)$ *is up, then* $S(r(1), \xi') = S(r(1), \tilde{\xi}')$*.*

Proof. The difference between ξ' and $\tilde{\xi}'$ is that the latter does not include events that (i) are incident to j and (ii) are not yet in ξ. Following that G is directed acyclic, the only such events that would affect S here are $\alpha_{r(1)}$ and its compliment. Since $r(1)$ was assumed up, the compliment is also assumed. Therefore, there are no more events that could be excluded by the use of $\tilde{\xi}'$ that affect the value of S.

□

Theorem 2 (Inductive SWS Solution with Incident Speed-Up Theorem)**.** *If* $j = d$*, then* $S(ij, \xi) = P(\bar{\alpha}_{ij} \mid \xi)$*. Otherwise,* $S(ij, \xi) = P(\bar{\alpha}_{ij} \mid \xi)\mathbf{E}_{\tilde{\xi}'} \sum_{k=1}^{g} S(r(k), \tilde{\xi}') \prod_{m=1}^{k-1} P(\alpha_{r(m)} \mid \tilde{\xi}')$*.*

Proof. This definition of S differs from the previous definition only in its treatment of incident vision. From the definition of the pathfinder's behavior, we know it will prefer to select $r(1)$ as its next step. This will be done exactly when the pathfinder does not see that $r(1)$ is down. If $g = 1$, then $r(1)$ will never be seen and will always be selected. Otherwise, $r(1)$ will be selected whenever it is up, which happens with probability $P(\bar{\alpha}_{r(1)} \mid \tilde{\xi}')$. This, however, is already included in the definition of the $r(1)$ subproblem, $S(r(1), \tilde{\xi}')$, so we need not express it explicitly here.

If $g = 2$, then $r(2)$ will never be seen, but will only be selected when $r(1)$ is down, which happens with probability $P(\alpha_{r(1)} \mid \tilde{\xi}')$. Similarly, in general, $r(g)$ will never be seen, but will only be selected when $r(1), r(2), \ldots, r(g-1)$ are all down, which happens with probability $\prod_{m=1}^{g-1} P(\alpha_{r(m)} \mid \tilde{\xi}')$.

Therefore, we may rewrite S as a sum following this total probability of how the pathfinder will behave next.

$\square$

Remark 1. *The latter definition of S eliminates the* $\max_{jk}$ *term and requires fewer iterations from the expected value operation, since, it is trivial to observe,* $2^h \left|\tilde{R}(\xi)\right| = |R(\xi)|$ *for some nonnegative integer h. Therefore, a speed-up of 2^h may be expected.*

Definition 6 (Beta Space)**.** *By $\mathbb{B}$ we mean the a finite space generated by a finite application of (i) $0, 1 \in \mathbb{B}$, (ii) $\beta_{ij} \in \mathbb{B}$ for all $ij \in E$, (iii) if $b \in \mathbb{B}$, then $1 - b \in \mathbb{B}$, (iv) if $b_1, b_2 \in \mathbb{B}$, then $b_1 b_2 \in \mathbb{B}$, and (v) if $b_1, b_2, b_3 \in \mathbb{B}$, then $b_1 b_2 + (1 - b_1) b_3 \in \mathbb{B}$. We note that (v) implies (iii) and (iv), but they are nonetheless included in this list for clarity.*

Lemma 2 (Expected Value in Beta Space)**.** *If $f(\xi) \in \mathbb{B}$, then $\mathbf{E}_{\xi'} f(\xi') \in \mathbb{B}$.*

Proof. We prove this inductively. When ξ' takes on a single realization, the expected value is equal to $f(\xi)$ without changes and is in $\mathbb{B}$ by assumption. For the general case, consider how the expected max is a sum weighted by a $P(\xi' \mid \xi)$ term and how this is equalivant to some $P(a_1, a_2, \dots) = P(a_1)P(a_2)\cdots = b_1 b_2 \dots$, where $a_1 \in \{\alpha_{ij}, \bar{\alpha}_{ij}\}$ for some ij. It follows that each $b_k \in \mathbb{B}$ and therefore $P(\xi' \mid \xi) \in \mathbb{B}$. With this in mind:

$$\mathbf{E}_{\xi'} f(\xi') =$$

$$\sum_{\xi'} f(\xi') P(\xi' \mid \xi) =$$

$$\sum_{a} f(a) P(a_1, a_2, \dots) =$$

$$\sum_{a} f(a) P(a_1) P(a_2) \cdots =$$

$$P(a_1)\sum_{a|a_1} f(a)P(a_2)\cdots + P(\bar{a}_1)\sum_{a|\bar{a}_1} f(a)P(a_2)\cdots =$$

$$P(a_1)\mathbf{E}_{a|a_1} f(a) + P(\bar{a}_1)\mathbf{E}_{a|\bar{a}_1} f(a) =$$

$$b_1 b_2 + (1 - b_1)b_3$$

This final step is made following the inductive assumption that smaller calls to the expected value are in $\mathbb{B}$; therefore, by assumption (v) of definition 6, $\mathbf{E}_{\xi'} f(\xi') \in \mathbb{B}$.

□

Theorem 3 (Range of SWS Solutions)**.** *For all ij and ξ, $S(ij, \xi) \in \mathbb{B}$.*

Proof. The value of S in the base case is $1 - \beta_{ij}$ for some ij, which is by definition in $\mathbb{B}$. For the general case, we can appeal to the previous lemma to reduce the solution to the multiplication of two terms, b_1 and b_2: the first of these follows the base case and is trivially in $\mathbb{B}$; the latter is the expected value of a function which, by inductive assumption here, is in $\mathbb{B}$. Therefore, the value of S too must always be in $\mathbb{B}$.

□

Remark 2. *Tighter bounds can be made during approximations, etc. by considering $\mathbb{B}$ as the range of S instead of the full interval* $[0, 1]$.

Appendix B

Code Used

```python
# dictionary of keys that returns the zero of the given type
class sparse_dok(dict):
    def __init__(self, t=bool):
        self.t = t
    def zero(self):
        return self.t(0)
    def __missing__(self, key):
        return self.zero()

# sparse_dok implementation of adjancency matrix
# also stores neighbors in/out as a linked list structure
class adjacency_matrix(sparse_dok):
    def __init__(self):
        self.t = bool
        self.before = []
        self.after = []
    def grow(self, k):
        if len(self.before) <= k:
            need = k-len(self.before) + 1
            self.before += [[] for x in range(need)]
        if len(self.after) <= k:
            need = k-len(self.after) + 1
            self.after += [[] for x in range(need)]
    def ins(self, j):
        self.grow(j)
        for i in self.before[j]:
            yield i
    def outs(self, i):
        self.grow(i)
        for j in self.after[i]:
            yield j
    def __setitem__(self, key, value):
        i, j = key
        self.grow(j)
        if not self[key]:
            self.after[i].append(j)
            self.before[j].append(i)
            super().__setitem__(key, value)

# similar to adjacency matrix, tracks vision
class sight_matrix(sparse_dok):
    def __init__(self):
        self.t = bool
        self.lines = []
    def grow(self, k):
        if len(self.lines) <= k:
            need = k-len(self.lines) + 1
            self.lines += [[] for x in range(need)]
    def vision(self, k):
        self.grow(k)
        for i, j in self.lines[k]:
            yield i, j
    def __setitem__(self, key, value):
        k, i, j = key
        self.grow(k)
        if not self[key]:
            self.lines[k].append((i, j))
            super().__setitem__(key, value)

# a sparse_dok, but it returns 1.0 for missing values
class state_matrix(sparse_dok):
    def __init__(self):
        self.t = float
    def zero(self):
        return 1.0

# base class for functions
class nonchainable(object):
    def __init__(self, func):
        self.func = func
    def __call__(self, *args):
        return self.func(*args)
    def __str__(self):
        return self.func.__name__

# base class for functions such that
# the range and domain are the same
class chainable(nonchainable):
    def __and__(self, other):
        def temp(*args):
            if len(args) == 1:
                args = (self(*args),)
            else:
                args = self(*args)
            return other(*args)
        temp.__name__ = "{0}_{1}".format(self, other)
        return type(self)(temp)

# we make the A and W here; the func
# that is decorated with @base_graph just
# needs to make edits
class base_graph(nonchainable):
    def __call__(self, N):
        A = adjacency_matrix()
        W = sight_matrix()
        G = (int(N), A, W)
        self.func(G)
        return G

# decorates a function that, given a G, makes edits
# to add edges
class grapher(chainable):
    def __call__(self, G):
        (N, A, W) = G
        G = (int(N), A, W)
        return self.func(G)

# similar to grapher, but for vision
class sighter(chainable):
    def __call__(self, G):
        (N, A, W) = G
        G = (int(N), A, W)
        return self.func(G)

# similar to grapher, but modifies
# a given bstat instead
class betastate(nonchainable):
    def __call__(self, G):
        (N, A, W) = G
        G = (int(N), A, W)
        bstat = state_matrix()
        self.func(G, bstat)
        return bstat

# similar to grapher, but returns
# a modified copy of the graph
class modder(chainable):
    def __call__(self, G):
        (N, A, W) = G
        G = (int(N), A, W)
        return self.func(G)

# a special sparse_dok for caching
# dynamic programming results
# defined by its encode/search system
class solver_cache(sparse_dok):
    def __init__(self):
        self.t = float
    def zero(self):
        return False
    def encode(self, S, I, path, G, bstat):
        return str((S, I, path, G, bstat))
    def search(self, encoding):
        return super().__getitem__(encoding)
    def __setitem__(self, key, value):
        S, I, path, G, bstat = key
        key = self.encode(S, I, path, G, bstat)
        super().__setitem__(key, value)
    def __getitem__(self, key):
        S, I, path, G, bstat = key
        key = self.encode(S, I, path, G, bstat)
        return self.search(key)

# decorator for a substater that, given
# several variables, yields possible visable substates
class substater(nonchainable):
    def __call__(self, bstat, path, G):
        for (substate, likelihood) in self.func(bstat, path, G):
            yield (substate, likelihood)

# decorator for a pruner
class pruner(nonchainable):
    def __call__(self, best, i, j, path, G, subbstat):
        return self.func(best, i, j, path, G, subbstat)

# constructor for a config
# note the reset_solver method
class config(object):
    def __init__(self, **cfg):
        self.cfg = {
            # number of vertices
            "N": 10,
            # minimally connected graph
            "base_graph": line_graph,
            # adds edges to base
            "grapher": random_graph(50),
            # adds vision to graph
            "sighter": random_vision(50),
            # risk-state of edges
            "betastate": coin_betastate,
            # encode based on visitability
            "solver_cache": visitable_cache,
            # yields substates of a subproblem
            "substater": what_ifs,
            # modifies the graph, usually making it simpler
            "modder": identity,
            # prunes unnecessary subproblems
            "pruner": apriori_prune,
            "solver": success # solves the problem
        }
        for (key, value) in cfg.items():
            self.cfg[key] = value
    def __getattr__(self, key):
        return self.cfg[key]
    def reset_solver(self):
        self.solver.cfg = self
        self.solver.cache = self.solver_cache()

# decorator for a solver
# defined by a func that returns a solution
# between 0 (bad) and 1 (good) and a boolean
# of whether we need to multiply it by the
# expected max solution of the next step
# this boolean return separates the config-touching
# logic of th substate iteration and the math-touching
# logic of the solver
class solver(nonchainable):
```

```
def __init__(self, func):
    self.func = func
    self.cache = None
    self.cfg = None
    self.top_call = True
def expected_max(self, i, path, G, bstat):
    (N, A, W) = G
    wsum = 0.0
    path.append(i)
    # substater does is not REQUIRED to explore the incident edges
    # allows for a 2^h to h speed-up
    for (subbstat, likelihood) in \
        self.cfg.substater(bstat, path, G):
        solutions = []
        best = 0.0
        for j in A.outs(i):
            if subbstat[i,j] < 1 and \
                self.cfg.pruner(best, i, j, path, G, subbstat):
                subsolution = \
                    self.cfg.solver(i, j, path, G, subbstat)
                solutions.append((subsolution, j))
                best = max(best, subsolution)

        solutions.sort(reverse=True)
        temp = 0.0
        choice = 1.0
        for subsolution, j in solutions:
            temp += choice * subsolution
            choice *= subbstat[i,j]
            if not W[i,i,j]:
                choice = 0.0
            if choice == 0.0:
                break

        wsum += likelihood*temp
    path.pop()
    return wsum
# I'm at s and know everything at s and along the path before
# path starts at 0 and ends with s
# I take i next; what is my probability of success?
# There are 2^h |V| ways to say I'm at s and know
# everything at s -- in a markovian system.
# There are deg(s) ways to say I take i next.
# Therefore, there are 2^h |V| deg(s) subproblems,
# or simpler, 2^h |E| -- in a markovian system.
```

```
    # h, here, is taken to be a small number representing the
    # maximum *new* information that will be seen upon entering
    # a any vertex.
    def __call__(self, s, i, path, G, bstat):
            format(self.func.__name__)
        if self.top_call:
            self.top_call = False
            result = self(s, i, path, G, bstat)
            self.top_call = True
            return (result, len(self.cache))
        else:
            result = self.cache[s, i, path, G, bstat]
            if result is not False:
                return result
            else:
                (value, done) = self.func(s, i, path, G, bstat)
                if not done:
                    value *= self.expected_max(i, path, G, bstat)
                self.cache[s, i, path, G, bstat] = value
                return value


from random import random, choice, shuffle

# a base graph to build on top of
# each vertex is connected to the one
# immediately after it
@base_graph
def line_graph(G):
    (N, A, W) = G
    for j in range(1, N):
        A[j-1,j] = True

# edges have density chance of being set,
def random_graph(density=50):
    def temp(G):
        (N, A, W) = G
        for i in range(N):
            for j in range(i+1, N):
                if random() < density/100:
                    A[i,j] = True

        return G

    temp.__name__ = "random_graph_{0}".format(density)
    return grapher(temp)

# lines of sight have lighting chance of being set
def random_vision(lighting=50):
    def temp(G):
        (N, A, W) = G
        for i in range(N):
            for j in range(i+1, N):
                if A[i,j]:
                    for k in range(i+1):
                        if random() < lighting/100:
                            W[k,i,j] = True

        return G

    temp.__name__ = "random_vision_{0}".format(lighting)

    return sighter(temp)

# Assume risk=50% everywhere
@betastate
def coin_betastate(G, bstat):
    (N, A, W) = G
    for i in range(N):
        for j in range(i+1, N):
            if A[i,j]:
                bstat[i,j] = 0.5


from collections import deque

# if we start at s, yield all edges
# assumes all are visitable
def bfsearch(G, s):
    (N, A, W) = G
    Q = deque()
    appended = [False for i in range(N)]
    Q.appendleft(s)
    appended[s] = True
    while Q:
        i = Q.pop()
        for j in A.outs(i):
            yield (i, j)
            if not appended[j]:
                Q.appendleft(j)
                appended[j] = True

# If I'm at S and I take I next, what all vertices
# do I have a chance to visit, including those two?
# let S be None to assume to reach I no matter what
def visitable_vertices(S, I, path, G, bstat):
    (N, A, W) = G
    Q = deque()
    visited = [False for i in range(N)]
    if S is not None:
        visited[S] = True
        if bstat[S,I] < 1:
            Q.appendleft(I)
            visited[I] = True
    else:
        Q.appendleft(I)
        visited[I] = True

    while Q:
        i = Q.pop()
        for j in A.outs(i):
            if bstat[i,j] < 1 and not visited[j]:
                Q.appendleft(j)
                visited[j] = True

    return visited

# solver_cache that uses an encoding based on visitability
class visitable_cache(solver_cache):
    # encoding based on what edges can still be visited
    def encode(self, S, I, path, G, bstat):
        (N, A, W) = G
        encoding = "{0}:{1}:".format(S, I) # where are we?
        visitable = visitable_vertices(S, I, path, G, bstat)
        if bstat[S,I] == 0: # encoding for first edge state
            encoding += "1"
        elif bstat[S,I] == 1:
            encoding += "0"
        else:
            encoding += "?"
        if visitable[I]: # if it isn't, who cares about the rest?
            for (i, j) in bfsearch(G, I):
                if visitable[i] and visitable[j]:
                    if bstat[i,j] == 0:
                        encoding += "1"
                    # i and j may both be visitable,
                    # but not the edge that connects them,
                    # so still have to check == 1 here
                    elif bstat[i,j] == 1:
                        encoding += "0"
                    else:
                        encoding += "?"
                else:
                    encoding += "0"

        return encoding


# bstat[i,j] == 0 when ij is up
#             == 1 when ij is down
#             == risk otherwise
# bstat is existing bstat or passed from parent call of what_ifs
# path is actually modified path from success
# note, bstat is actually just share memory between calls
@substater
def what_ifs(bstat, path, G):
    (N, A, W) = G
    s = path[-1]
    visitable = visitable_vertices(None, s, path, G, bstat)
    found = None # check everything we can see for something to change
    for (i, j) in W.vision(s): #bfsearch(G, s):
        if i != s: # incident vision is handled by expected max
            if visitable[i] and visitable[j]:
                if bstat[i,j] not in [1, 0]:
                    found = (i, j)
                    break

    # base case, no seen and unassumed edges found, so just yield
    # when we find something, put it up, down, then back to normal
    if found is not None:
        i, j = found
        risk = bstat[i,j]
        bstat[i,j] = 0
        for (subbstat, likelihood) in what_ifs(bstat, path, G):
            yield (subbstat, likelihood*(1-risk))

        bstat[i,j] = 1
        for (subbstat, likelihood) in what_ifs(bstat, path, G):
            yield (subbstat, likelihood*risk)
        bstat[i,j] = risk
    else:
        yield (bstat, 1.0)

# don't make any modifications to the graph
@modder
def identity(G):
    return G

# if the current best is better than even crossing i to j, then
# don't bother computing success from i through j to d.
@pruner
def apriori_prune(best, i, j, path, G, bstat):
    (N, A, W) = G
    return W[i,i,j] or best < 1-bstat[i,j]

# true solution
# simply, the safety and whether we are done
@solver
def success(s, i, path, G, bstat):
    (N, A, W) = G
    sfty = 1-bstat[s,i]
    if i == N-1 or sfty == 0.0: # base case, no "later" to worry about
        return (sfty, True)
    else:
        return (sfty, False)

# handler for running a single trial
def trial(cfg, path, G, bstat, reset=True):
    cfg = config(**cfg)
    if reset:
        cfg.reset_solver()

    (N, A, W) = G
    likelihood = [float("-inf") for i in range(N)]
    s = path[-1]
    runtime = 0
    for i in A.outs(s):
        succ, runtime = cfg.solver(s, i, path, G, bstat)
        likelihood[i] = succ

    return likelihood, runtime

# random graph, except each vertex can only connect to
# and see the x after it
# x=0 adds no edges
# x=-1 all edges
# x=-2 adds all but the longest edge
def neighborhood_graph(dist=3, density=50):
    def temp(G):
        (N, A, W) = G
        x = dist
        if x < 0:
            x = N+x

        for i in range(N):
            for j in range(i+1, min(i+x, N)):
                if random() < density/100:
                    A[i,j] = True

        return G

    temp.__name__ = "neighborhood_graph_{0}_{1}".format(dist, density)
    return grapher(temp)

# each vertex can see its incident edges
@sighter
def incident_vision(G):
    (N, A, W) = G
    for i in range(N):
        for j in range(i+1, N):
            if A[i,j]:
                W[i,i,j] = True

    return G

# modder that applies noise to the edges at a given rate
def noiseEdges(rate):
    def temp(G):
        (N, A, W) = G
        B = adjacency_matrix()
        V = sight_matrix()
        for i in range(N):
            for j in A.outs(i):
                k = j
                if random() < rate/100:
                    candidates = \
                        [x for x in range(i+1, N) \
                        if not A[i,x] and not B[i,x]]
                    if candidates:
                        k = choice(candidates)

                B[i,k] = True
                for m in range(j):
                    if W[m,i,j]:
                        V[m,i,k] = True

        G = (N, B, V)
        return G
    temp.__name__ = "noiseEdges_{0}".format(rate)
    return grapher(temp)

# like noiseEdges, for for lines-of-sight
def noiseLines(rate):
    def temp(G):
        (N, A, W) = G
        V = sight_matrix()
        for i in range(N):
            for j, k in W.vision(i):
                n, m = j, k
                if random() < rate/100:
                    candidates = [(x, y) for (x, y) in bfsearch(G, i) \
                                  if not W[i,x,y] and not V[i,x,y]]
                    if candidates:
                        n, m = choice(candidates)

                V[i,n,m] = True

        G = (N, A, V)
        return G
    temp.__name__ = "noiseLines_{0}".format(rate)
    return sighter(temp)

# a modder that removes all vision from the graph
@modder
def remove_all_vision(G):
    (N, A, W) = G
    V = sight_matrix()
    H = (N, A, V)
    return H

# a modder that removes all but incident vision from the graph
@modder
def incident_vision_only(G):
    (N, A, W) = G
    V = sight_matrix()
    for k in range(N):
        for j in range(N):
            if W[k,k,j]:
                V[k,k,j] = True

    H = (N, A, V)
    return H

# a modder that adds effective vision to the graph
@modder
def effective_vision(G):
    (N, A, W) = G
    V = sight_matrix()
    for j in range(N):
        counts = {}
        num_parents = 0
        for k, m in W.vision(j): # copy all from original j
            V[j,k,m] = True

        for i in A.ins(j): # copy all shared by its new parents
            num_parents += 1
            for k, m in V.vision(i): # note the V here and not W
                if k >= j:
                    if (k, m) not in counts:
                        counts[k,m] = 0

                    counts[k,m] += 1

        for k, m in counts:
            if counts[k,m] == num_parents:
                V[j,k,m] = True

    H = (N, A, V)
    return H

# solver for hammy down
def hammy_down_substater(a):
    def temp(bstat, path, G):
        (N, A, W) = G
        found = None
        in_child = None
        s = path[-1]
        visitable = visitable_vertices(None, s, path, G, bstat)
```

```
# so here's the thing with the visitability
# in the what_if and hammy substaters.
# when we set a nearby edge to down, and that
# prevents us from even getting
# to some other edge we can see, who cares what
#the up/down of that unreachable
# edge is? once we set the nearby edge back up,
# then we'll care.
for i in path:
    for j, k in W.vision(i): # first, turn off in parents
        if j != i: # incident vision is handled by expected max
            if visitable[j] and visitable[k]:
                if i == s:   # then, turn on here
                    if bstat[j,k] not in [1, 0]:
                        found = (j, k)
                        in_child = True
                        break
                else:
                    if not W[s,j,k] and bstat[j,k] in [1, 0]:
                        found = (j, k)
                        in_child = False
                        break

    if found is not None: # needed to break out of two loops
        break

if found is not None:
    if in_child:
        risk = bstat[j,k]
        bstat[j,k] = 0
        for (subbstat, likelihood) in temp(bstat, path, G):
            yield (subbstat, likelihood*(1-risk))

        bstat[j,k] = 1
        for (subbstat, likelihood) in temp(bstat, path, G):
            yield (subbstat, likelihood*risk)

        bstat[j,k] = risk
    else:
        risk = bstat[j,k]
        bstat[j,k] = a/100
        for (subbstat, likelihood) in temp(bstat, path, G):
            yield (subbstat, likelihood)

        bstat[j,k] = risk
        else:
            # base case, no changes to make, so just yield
            yield (bstat, 1.0)

    temp.__name__ = "hammy_down_substater_{0}".format(a)
    return substater(temp)

# a "solver" for the semirandom walk
@solver
def guesser(s, i, path, G, bstat):
    (N, A, W) = G
    sfty = 1-bstat[s,i]
    if sfty == 0.0:
        return (0, True)
    else:
        return (random(), True)

# helper for the main-ish area below
def make_graph_bstat(cfg):
    cfg = config(**cfg)
    draw_config_table(cfg)
    G = cfg.base_graph(cfg.N)
    G = cfg.grapher(G)
    G = cfg.sighter(G)
    bstat = cfg.betastate(G)
    return G, bstat

# helper for the main-ish area below
def modify_graph(G, cfg):
    cfg = config(**cfg)
    return cfg.modder(G)

# helper for the main-ish area below
def traverse(G, cfg, scheme, bstat, real_bstat):
    (N, A, W) = G
    # start at source
    path = [0]
    # build config for the trial by combining default and scheme configs
    trial_cfg = {}
    trial_cfg.update(cfg)
    trial_cfg.update(scheme)
    # modify how the solver wants the graph, if needed
    H = modify_graph(G, trial_cfg)
    copy_bstat = state_matrix()
    # initialize "seen" risk-states
    for i, j in bfsearch(G, path[-1]):
        copy_bstat[i,j] = bstat[i,j]

    runtimes = []
    while True:
        # update risk-states with new information
        for j, k in W.vision(path[-1]):
            copy_bstat[j,k] = real_bstat[j,k]

        # solve the problem, starting from scratch
        # at a further point with new info
        # only reset the solver at the first step
        # big speed up, no adverse effects
        likelihood, runtime = trial(trial_cfg, path, H, \
            copy_bstat, path[-1]==0)
        runtimes.append(runtime)

        # select the best next step,
        # prefering higher indices in case of ties
        choice, to_beat = -1, 0
        for (i, score) in enumerate(likelihood):
            if score >= to_beat:
                choice, to_beat = i, score

        # if the choice was bad, give up, return false
        # elif the choice let us win, we're done, return true
        # else, just update the path and continue
        ij = (path[-1], choice)
        path.append(choice)
        if choice == -1:
            return (False, runtimes, path)
        elif to_beat <= 0:
            return (False, runtimes, path)
        elif not A[ij]:
            return (False, runtimes, path)
        elif real_bstat[ij] == 1:
            return (False, runtimes, path)
        elif choice == N-1:
            return (True, runtimes, path)

# create Q random graphs each with N vertices
# for each, run T oracle-positive trials
def verify(Q, T, N, **schemes):
    # starting configuration
    cfg = {"N": N,
            "betastate": fixed_betastate(25),
            "base_graph": line_graph,
            "grapher": neighborhood_graph(4, 100) & noiseEdges(5),
            "sighter": incident_vision & noiseLines(25)
          }

    print("{0}\t{1}\t{2}\t{3}".format(Q, T, N, len(schemes)))
    # collect the names of the scheme arguments
    scheme_names = [k for k in schemes.keys()]
    # start histograms
    hists = {scheme_name: [0 for t in range(T+1)] \
        for scheme_name in scheme_names}
    for q in range(Q):
        # create a random graph based on the configuration
        G, bstat = make_graph_bstat(cfg)
        (N, A, W) = G

        num_succeed = {scheme_name: 0 \
            for scheme_name in scheme_names} # start counts
        t = 0
        while t < T:
            # realize the risk-state for the edges,
            # but keep this hidden from the solvers
            real_bstat = state_matrix()
            for i, j in bfsearch(G, 0):
                if A[i,j] and random() < bstat[i,j]:
                    real_bstat[i,j] = 1
                else:
                    real_bstat[i,j] = 0

            # first check the oracle, using the default cfg and
            # giving it all the info up front
            success, runtimes, path = \
                traverse(G, cfg, cfg, real_bstat, real_bstat)
            if success:
                t += 1

                # only attempt the approximations
                # if the oracle could find a solution
                # (i.e., one exists)
                shuffle(scheme_names)

                # if the scheme can traverse all the
                # way to the end, give it a tally
                for scheme_name in scheme_names:
                    scheme = schemes[scheme_name]
                    success, runtimes, path = \
                        traverse(G, cfg, scheme, bstat, real_bstat)
                    if success:
                        num_succeed[scheme_name] += 1
                    print("{0}\t{1}\t{2}\t{3}\t{4}".format(\
                        scheme_name, len(runtimes),\
                        "\t".join(map(str, runtimes)),\
                        "\t".join(map(str, path)),\
                        success))

        # update hists and output
        for scheme_name in scheme_names:
            hist = hists[scheme_name]
            hist[num_succeed[scheme_name]] += 1
            print("{0}\t{1}".format(scheme_name,\
                "\t".join(map(str, hist))))

        print("\n------------------\n")

def main():
    verify(100, 1, 100, # Q, T, N
        random={
            "solver": guesser
        },
        incident={
            "modder": effective_vision & incident_vision_only
        },
        hammy={
            "substater": hammy_down_substater(50),
            "modder": effective_vision
        },
        quick={
            "modder": remove_all_vision
        }
    )
```