The Art Of Being Efficiently Lazy

1. The “I Don’t Want to Do This” Principle

I have something that I need to admit: when I play video games, I am very lazy.

If I am joining Minecraft or Roblox and I need to get from point A to B, I don’t want to calculate coordinates, dodge every obstacle, or carefully plan a route. I just want to click on the map and have my character magically know where to go. But it turns out that computers actually feel the same way as I do.

If a computer tried to check every possible path though a map to find the best one, it would run into a problem called combinatorial explosion, which essentially means that your computer will crash due to the amount of work. At every step, there are multiple choices, left, right, forward, and those choices multiply with each step you take. Even in a simple 50x50 grid, the number of possible paths becomes larger than the number of atoms in the known universe! What I am trying to say is that trying every option isn't just slow, it is basically impossible.

So instead of working harder, computers work smarter. They use math to their advantage to avoid unnecessary effort and jump straight to the best solution. In other words, they’ve mastered the art of being efficiently lazy.

2. Dijkstra’s “Bucket of Paint”

To make sense of a complicated map, mathematicians will use Graph Theory to help them.

In this framework, the word is simplified so much. Actually, only into two parts: nodes (points, like positions on a map), and edges (connections between those points). This lets us turn something messy, like a forest, into something structured and mathematical.

One of the most classic ways to find the shortest path in a graph is Dijkstra's algorithm. Imagine you are trapped in a completely dark maze with a bucket of glowing paint. You pour the paint onto the floor and it spreads in every direction outwards at the same time, though every single corridor, corner, and dead end. At one point, the point will reach the exit, and the path it took to get there first must be the shortest possible path.

The big idea behind this algorithm is that it explores all possible directions, but always expands outwards from the closest known points first. It will then keep track of the costs of reaching each location and build outwards step by step.

Now, real video games are not always made of identical paths. Some areas are harder to cross than others, which is where weighted graphs come in. Each step has a cost, for example:

- a smooth marble floor = cost 1
- thick mud = cost 5
- and walls = cost infinity (which makes them impossible to cross as the algorithm will register it as too much of a cost.)

But let's say you start moving through the maze. At first, your cost will be 0 at your starting position, and from there, it will explore nearby squares, adding the correlating cost to where you moved to your total. As the algorithm spreads outwards, it keeps updating the cheapest known cost to reach each square. If it finds a cheaper way to get somewhere, it will replace the old value.

Eventually, it will reach the exit, and since it always builds up from the lowest-cost paths first, the route that it found is guaranteed to be the cheapest. The issue with this algorithm though is it actually doesn't know where the exit is. It treats every direction equally, so even if the exit is straight ahead, it will still spend its time checking the paths that are behind or to the side. This makes it careful and reliable, but not efficient enough.

3. The A* Method

To fix the inefficiency faced by the algorithm, the A* search algorithm is used. It uses Dijkstra's idea, but adds the improvement of giving the algorithm a sense of direction. Instead of only tracking how far it has already traveled, A* will also estimate how far it has to go, so it doesn't check paths that are very far from the exit. The estimate is called a heuristic, which is a kind of mathematical "guided guess."

The formula used is:

$f(n) = g(n) + h(n)$

$g(n)$ is the exact cost from the start to the current square it is on

$h(n)$ is the estimated cost from the current square to the exit or goal

$f(n)$ is the total score, which is the sum of $g(n)$ and $h(n)$

The algorithm will always choose to explore the square with the lowest f(n), rather than every square. Now to connect it back to the maze from section 2, imagine you are standing at the start and you have two possible moves:

Option 1:

- Step onto marble so the cost so far is g = 1
- You are still far from the exit, with an estimated cost of h=10
- The total is f = 1 + 10 = 11

Option 2:

- Step onto mud so the cost so far is g = 5
- You are close to the exit with an estimated cost of h = 3
- The total is f = 5 + 3 = 8

Even though the mud is actually more expensive to step on, A* will choose Option 2 because overall, it looks more promising. This is what makes the A* search so powerful, as it doesn't think locally (of what is the cheapest now?), but instead thinks ahead of what looks more promising.

Unlike Dijkstra's "paint" which spreads everywhere, A* is more like a guided flow, where it leans towards the exit while still ensuring that its path is optimal. As long as the heuristic is chosen carefully (such as estimated distance in a realistic way), A* will still find the shortest path, but much faster, as it will ignore large areas of the math that aren't needed. This shows efficient laziness at its very peak, doing less work, but thinking very intelligently.

4. Why This Actually Matters (Beyond Games)

It isn't just video games that use this algorithm, but navigation systems, like Google Maps, use similar algorithms to find the fastest route through traffic. They don't test every possible road, but instead they estimate which directions are most promising and focus on directing you to those. Behind the scenes, the map is treated like a graph, where the intersections are looked at as nodes, and the roads are edges with different costs based on the distance, speed, and real-time traffic. If a road becomes too busy, its cost increases, which will register in the algorithm and will be less likely to be chosen. This system is also constantly updating the values and re-evaluating routes as conditions change, which is why it can reroute you if there is an accident or delay ahead.

5. The Art of Being Efficiently Lazy

So, maybe being “lazy” isn't actually a bad thing. When I don’t want to manually navigate every step in a game, I’m really asking the same question that mathematicians asked, “What is the smartest way to avoid doing unnecessary work?” Math, the A* search method, don't just solve the problem, but solve it in an efficient way. And in the end, it means one thing: I don’t have to do the work, but math does it for me.