Dire Straits & creating random RM maps

Table of Contents

Introduction

Many sections of this describe programming. If you're unfamiliar with that, just read 'A Non-Technical Summary' and skip the rest. Although Dire Straits is specifically built for AoE2:DE, the map generation process is completely separate from the game. That description starts at 'Overall Creation Pipeline.'

To make things work in AoE2, you'll need to know how RMS (random map scripting) works (an overview of RMS is available here). The approach I'm going to describe involves a lot of programming outside of the RMS framework. It approximates a random map generated by a separate program (which isn't held back by AoE2's limitations) by translating each instance (or seed) of the map to the appropriate direct_placement calls. The RMS then selects one of these at random and completes the maps in-engine by adding elevation, terrains, and objects. While we don't have enough RMS capacity to make this process perfect, we do have enough to greatly expand the range of maps we can make.

However, depending on how complicated it ends up being, doing so may not be practical for someone who just wants to make a few maps. Dire Straits demonstrates that it's possible and realistic, but I probably would not have spent a couple hundred hours on it just to make one map if I wasn't also exploring the limits of what is possible and experimenting with random generation for other reasons. A gameplay overview of Dire Straits is available here. The Python & RMS code used to build it is available here.

In particular, you don't have to do everything from scratch for a map with water that works for all team sizes like I did. If you can utilize a versatile software library for the task (I don't know of one. I've provided all my code, but it's more of an example than a library) or pick a much simpler map to design, the time to build a new map could be reduced to under 10 hours. Maybe you can find a way to utilize map generation from other games or non-gaming software. I'm sharing everything I know about one way to do it, but it's not the only way. The main point is that the concept itself has potential.

A Non-Technical Summary

The typical AoE2 random map tells the program what to make, and then makes something similar every time. Commands which yield more varied maps often lead to severe balance issues when water is involved. People have tried to "fix" the imbalance by removing as much randomness as possible, sometimes to the point where a map basically becomes a scenario. Doing that undermines much of what makes the Random Map game mode special and "random." It also doesn't improve balance for most game settings. When the range of possible outcomes is narrow, it becomes harder to defeat the civilizations which have the best bonuses for that specific situation because that specific situation comes up 100% of the time.

Dire Straits generates the basic map geography (the shallow straits, lake positions, and town center locations) ahead of time using a separate program. Instead of being told exactly what to make, it makes infinite* maps and then filters through them to find the best ones according to various criteria. As a result, it's able to generate balanced maps that cover a larger portion of the possible designs. However, things like elevation, forests, and placing resources are handled by AoE2 itself. (*This is an approximation/oversimplification of the process.)

The straits meander naturally and randomly, with the goal of emulating a real-world river. I added restrictions to avoid the map edges, turn gently, and trend away from places already visited. Between 3 and 6 lakes are placed along the river in random locations, but with some minimum separation between them. Player starting positions are guaranteed to be not too close to water, but not too far from fish. The exact numbers vary based on map size.

Despite player arrangement being random, the majority of maps involve one team occupying one half of the map and the other team occupying the other half. The majority of maps also involve centrally-located waterways with one team on each side. Some of the arrangements are even symmetric, or close to it. These were not part of the program design, but were emergent characteristics based on the various balance criteria.

At times the balance is asymmetric and one team has slightly easier access to the land while the other team has slightly easier access to the water, but since all the land is connected and all the water is connected this shouldn't be a problem. Although each player is guaranteed to be close to some fish, how important the water will be will vary from map-to-map and player-to-player. Sometimes there are more lakes than players (always the case in 1v1). Sometimes there are more players than lakes (always the case in 4v4). Coordination between teammates will be helpful.

The resources are predictable enough so that everyone can execute a basic strategy, but I wanted there to be a larger skill gap between a basic build and scouting to find the optimal plan. There's some untapped potential for making the resources more varied, but unfortunately the civilization balance is so fragile and linked to exact quantities of resources that it's unclear how this should be implemented.

This map was a proof of concept for a single design that can be built with this method. I picked something (mixed maps with both land and water) that has always been challenging to make if we wanted unpredictability, variety, asymmetry, and fairness. There are countless more possibilities, some of which can yield maps which are nothing like what we've played.

Terminology: Map, Random, Symmetric, Balanced

When I refer to a map, I'm referring to the geography that the RMS tends to generate, not the resources. I will try to use more specific language when referring to a specific instance of a map. The RMS toolset gives you pretty decent options for placing resources. Its shortcomings largely come from being unable to control or validate too much of the geography (primarily <LAND_GENERATION> in the RMS), and that's where this guide comes into play.

Randomness is not a boolean property. In this context, it's a measure of how unpredictable the map (geography) is. There's a significant gradient between "not random" (or predictable) and "extremely random." When it comes to AoE2, even the most random maps that players will tolerate are very low on the overall scale of randomness, in an absolute sense.

Every AoE2 RMS that places elevation, forests, and resources in nondeterministic locations will have some amount of randomness, but if the overall geography is basically the same every time, then that map is categorized as non-random. That means its level of variety and unpredictability falls on the lower side compared to other AoE2 maps. Collections of maps (eg. Random Land Map, Full Random, Ultra/Mega Random) complicate this analysis, and are mostly a distraction from the task. I'm just treating each map individually.

One purpose of randomness in strategy game map design is to put players in novel situations that encourage exploration and adaptation, which is the defining characteristic of the "Random Map" game mode and Age of Empires' multiplayer. (This includes games versus the AI. It excludes things like campaigns.) This variety gives the game more replayability. The unpredictability plays well with the fog of war mechanics involving both hidden information and imperfect information to give the game depth. (Having a large quantity of non-random maps gives the game breadth, which is separate from depth. But like the topic of map collections, it's a distraction from the goal of making a single random map.)

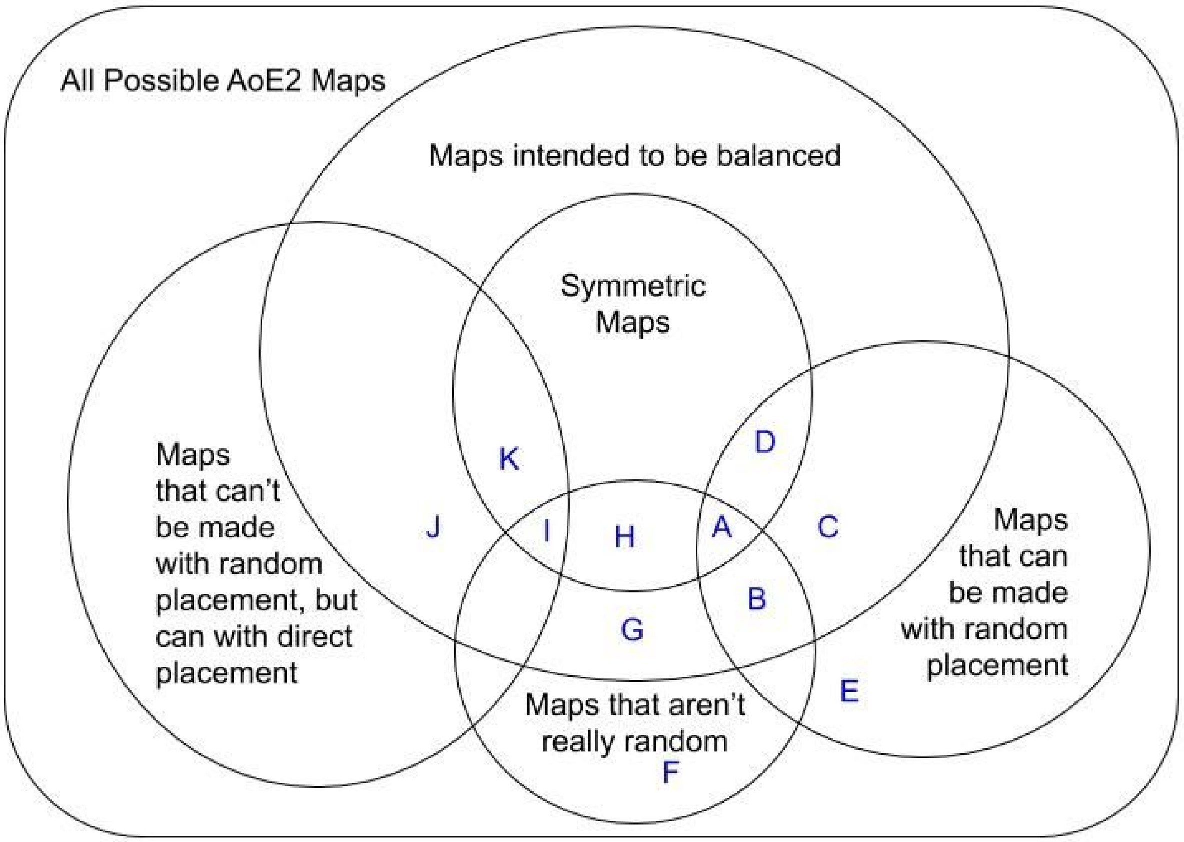
Like randomness, symmetry is also non-boolean. When I call something symmetric that doesn't mean everything is perfectly mirrored, but rather that it has a relatively high amount of symmetry compared to other maps. However, symmetry and randomness are not mutually exclusive.

Knowing that a map is symmetric doesn't give players as much information as you might think. Even in the extreme scenario of a completely mirrored map, half the map is still unexplored, and fog of war still exists. AoE2 is primarily an economic game, and the uncertainty of what your map will be like is a more significant source of randomness than the ways in which your opponent's map differs from your own. Although many non-random maps are symmetric, all symmetric maps don't have to be predictable. (The following section has some examples related to this.)

Balance in AoE2 involves having the outcome of a game determined by player decisions and actions rather than by things outside their control. Fairness of resources is essential for balance, and when you don't know which civilizations players are playing, forcing symmetry of resources is the most reliable way to accomplish that. However, high map symmetry and full map predictability are not requirements for balance. Forcing those is merely one of the easier ways to achieve it (because the capabilities of the RMS engine are limited).

Assuming symmetric resources, all symmetric maps reach a minimum threshold of balance, whereas only some asymmetric maps meet or exceed that level. I expect that wildly asymmetric maps have a higher chance of being imbalanced, but how you want to control that is up to you. With Dire Straits, I didn't try to influence symmetry directly. I just included some logic to filter out imbalanced maps and was comfortable with the result where some maps have high symmetry, most have slight asymmetry, and only a few have moderate asymmetry. That's easier said than done, so it might be easier to lean towards high symmetry. The approach for map generation outlined here cares more about achieving randomness and should be compatible with whatever level of symmetry you choose.

The design directions of AoE2 maps


All Possible AoE2 Maps
Maps intended to be balanced
Symmetric Maps
Maps that can't be made with random placement, but can with direct placement
Maps that can be made with random placement
Maps that aren't really random
A
B
C
D
E
F
G
H
I
J
K

- A: many map concepts, and an alarmingly high percentage of custom multiplayer-oriented maps
    - Acropolis, Arabia teamgames, Arena, Four Lakes, Golden Swamp, Mediterranean, Oasis
- B: some maps, especially with lower player counts
    - Arabia 1v1s, Black Forest (especially with ponds), Islands 1v1s, Migration
    - There isn't always a clear separation between Group A & Group B in general (it's somewhat subjective), but the RMS engine tends to yield higher symmetry with higher player counts
- C: few maps
    - Archipelago, Coastal, Nomad, Yucatan
    - These tend to generate things randomly by giving loose parameters to the RMS engine, whereas maps in Group A pick strict parameters which basically coerce the engine into making the same thing every time
    - They're intended to be fair, but tend not to have the safeguards to guarantee balance
- D: a rare category
    - Scandinavia
    - You could argue that some of these belong in Group A or some of Group A belongs here.
- E: some maps, often less well-known because people prefer to play balanced maps instead
    - Salt Marsh
- F: real world maps
- G & H: I assume some scenarios and ZR maps belong here
- I: few maps, generally with just 1 layout and relatively new (because they rely on newer features)
    - Cup and Bay from Hidden Cup
    - Examples I've seen have much less randomness than maps in group A. Maybe the diagram could indicate that, but it's complicated enough
- J: this guide describes how to make these
    - Dire Straits
- K: this guide describes how to make these

As examples to make things easier to parse, I listed maps that I thought would be the most familiar among the originals included on the disc and whatever was in the DE ranked ladder map pool when this document was written. I'm aware that custom maps are better than many of these, but there are too many of those to categorize, even if it was straightforward to find the most recent or representative versions of them.

Despite 20 years of RMS scripting, the maps we have made only cover a small fraction of the possibility space of all random map designs. There are good reasons for that, as the RMS engine isn't very sophisticated and optimizing for balance tends to lead people down the path of making non-random maps in the RMS framework instead of making actual random maps. Sometimes, this isn't even intentional, and it can happen gradually as successive updates to a map keep reducing its variety until it's basically a scenario with minor deviation introduced by elevation and resource spawns. Players might appear to spawn in a wide range of places, but if the enemies are always in the same place (relative to the center), that often doesn't add any actual strategic variance. Map collections don't really fix the issue since their randomness is still just inherited from the underlying maps.

The core structure of the Random Map mode rewards a wide variety of skills, one of which involves coming up with tactics and strategies for unexpected situations. This involves scouting, analyzing the map, figuring out what your opponent knows, and combining that with strengths & weaknesses of civilizations and of players. Intentional or not, removing asymmetry and randomness ends up devaluing those skills. There's no shortage of maps which do that, but nowhere near enough balanced maps which don't.

Combining direct_placement with offline out-of-engine map generation and validation is one way to expand into some of the underexplored design space (groups J & K in the diagram), particularly for types of maps that are difficult to balance without resorting to generating the same thing every single time. The rest of the guide describes some methods for accomplishing that.

The approach has benefits which can enhance existing designs as well. Programs can try lots of different configurations and pick the best ones whereas RMS-based maps have to generate something acceptable in the first attempt. If you can identify that a map is imbalanced whenever some situation occurs, and that situation occurs 10% of the time, all you need to do is generate 11% more maps and filter out the ones where that situation occurs. With programming this is easy. With RMS, it may be impossible. Even if you can figure out how to prevent that situation from ever happening, doing that might also prevent a lot of completely valid and acceptable maps from being generated.

Additionally, you aren't constrained too much by what you can accomplish if you aren't relying on RMS commands for everything. Even when they work correctly, there's only so much they can do. You have a lot more freedom when you can do anything that's possible with programming. However, for making the maps function in AoE2, there are some restrictions to keep in mind, which are described in the next section.

Limitations of RMS and direct_placement

create_land limit

The game only supports 99 create_land calls. That's sufficient for making a lot of different types of maps, but the resolution is kind of low once you start looking at larger player counts and map sizes. Exporting an entire map instance tile-by-tile is not viable.

To use this for complicated things (for example, spawning a lengthy meandering river), you can use some of this space to lay down pieces of the intended design at enough intervals, and then let a painting process in terrain_generation finish the job. For Dire Straits, there were enough commands in land_generation to generate the straits completely for tiny maps, but

I could only provide an approximation of them for larger sizes, so I had to do some complicated stuff in terrain_generation to complete them.

RMS length limit

There isn't a hard limit (that I know of), but it takes a long time for AoE2 to parse the RMS file. If you want to export thousands of instances, it will add dozens of seconds to the loading time in-game. The game doesn't show a progress bar or anything and just becomes unresponsive while it's loading the map (but it should load eventually without crashing), so just be aware of how large the resulting script is. You should only export the unique parts of each map instance (eg. the land generation) in order to save space and not duplicate code. Even if none of the duplicate code gets executed, it still seems to get parsed.

This creates an incentive to make each map instance you want to export as small as possible. If you want to make a map that works for 1v1, 2v2, 3v3, and 4v4, you can increase the number of instances you export if you can use the same set of commands for each team size. This is what I did with Dire Straits (mostly because it's a tech demo), and it made the programming much more complicated. If you don't do this, you get to use completely different maps for different team sizes instead of having to share them, which is a good thing for balance and variety. However, you run out of space more quickly.

The reason for wanting lots of instances is to make the map impossible to memorize. Since the game is about exploring, someone shouldn't be able to deduce the entire map just from observing a small portion of it. However, think about how much time someone would have to spend on a map before this happens. I mainly wanted to export lots of instances for this map as a proof of concept. I designed the program expecting to use a few thousand, but the loading times were exceeding 20 seconds so I settled on 360 at the end (which is more like 1440 since each instance supports each team size).

If someone plays a map many times and ends up playing on the same instance twice, it's not that bad, presumably because they already got many hours of enjoyment from it. If elevation, forests, and resource locations are generated randomly, they might not even notice. There are ways to generate fresh instances nobody has seen before if that's necessary for some ultra-competitive matches or something.

There may be ways to circumvent this limit with RMS's include functionality, but I'm not sure how much that would improve things. It would make the map difficult to share for multiplayer play. There are ways to sort of generate more map instances than you actually export. For one of those methods, see 'Increasing Randomness with Looped Straits.'

precision limit

Direct placement specifies coordinate values as a whole number percentage. Thus, the distance between two points depends on the map size. This poses some challenges for making the same instance work for all team sizes, so that's yet another reason to avoid doing it that way.

If you want the most precision, I'd recommend generating maps using the percentage coordinate system, and then validating whether or not they're good enough by translating them to the actual map size. Dire Straits generates maps on a 101x101 grid. It might be better to use 100x100 or 99x99 or 101x100 or 100x99 if you can figure out exactly what the game engine is doing. My map didn't do anything near the edges, so it didn't matter for me, but it's something to be aware of. Also pay attention to whether X Y specifies the center of the region you're representing or one of its corners (this matters more on larger map sizes).

The resolution of 1 percent shouldn't be a huge limitation for generating general map geography, but it probably precludes you from doing extremely precise things for other situations. It's mostly something to keep in mind if you want to keep areas separated by a certain amount of tiles. For example, in Dire Straits the difference between player's town center and the fish is larger on larger map sizes because I only guarantee a certain quantity of separation as a percentage.

Other details:

- Because of rounding, the distance between two percentage values may vary by 1 tile as those coordinates are translated with different offsets based on how exactly the percentages get translated to specific tiles.
- Town centers (and probably other buildings) specify their position as the bottom of the building instead of the middle. It might be a closer approximation to translate player positions by [-1, 1] on tiny maps. I think it's mostly just for appearance since villagers don't really exit from the topmost tile of the TC, but I'm not 100% sure about this.
- Be careful at the very edges of maps with position values of 100 (and possibly 0). I believe RMS has some bugs involving some of those coordinates.

direct_placement, land_generation, and variables

If you follow the basic pattern of keeping all the unique stuff for a map instance in land_generation, and then sharing the rest of the RMS for all instances, then you can minimize the cost (in terms of loading time) of including more instances. A few thousand lines of complicated and branching terrain generation might seem crazy, but it's not a lot compared to 200 lines of land_generation multiplied by 500 instances.

If you want to control the locations of components beyond land_generation commands, it's not very straightforward. But you can generate terrains and land_ids and zone numbers and base_elevation and then reference those values later on in the script. That may or may not be sufficient for what you want to do. For example, you can create lands in certain locations and then only spawn elevation on those terrains.

Another thing you can do is define boolean variables. If you want to use certain logic for some map instances but not others, you can define certain booleans in some of the instances and not others, and then refer to them later. (Alternatively, you could export the commands you want to run if they're unique to each instance, but that would make the script larger, and therefore slower, or it would force it to follow a certain structure.) In Dire Straits, I specified a few variables. One indicated whether extra hills would be appropriate for the map (based on how much land area there was). Another indicated what the base elevation should be, based on how close any player was to a strait.

Overall Creation Pipeline

Dire Straits approximately follows this process to create maps:

1. Generate many meandering rivers (calculate_routes.py)
2. Precompute random patterns of spaced-apart lakes and spaced-apart players (calculate_stencils.py & precalculate_lake_grids.py)
    a. It has almost every single possible configuration for the lakes
    b. It does not have every single possible configuration for the players, but it has a lot of them
    c. Lakes and players are computed separately and independently. This lets each pattern have more than 8 players so that the program can find a subset of 8 that's valid based on keeping players close to lakes (but not too close) and keeping the river out of their base. More information can be found in 'Some implementation Details.'
    d. When I refer to going through all the combinations, I'm referring to looping through each possible intersection of each of these patterns. It could actually be all possible combinations if you use a small enough coordinate space and devote enough compute resources to this task. That's difficult for Dire Straits because it's solving for two dependent variables at once (lake positions & player positions). In general, I'd recommend only solving for player positions and treating everything else as fixed as if it's part of the geography generation of Step 1.
3. For each river, examine all possible combinations of lakes & player positions (precalculate_route_info.py & calculate_layouts.py & choose_layouts.py). Filter out the ones that don't work for any of the following reasons:
    a. They don't have space for enough lakes
    b. They don't have space for 8 players

   c. They won’t fit within 99 create_land calls
   d. They aren’t balanced enough
      i. This checks each map size (1v1, 2v2, 3v3, and 4v4) individually
      ii. Considerations include players having similar access to land, access to fish, and access to teammates
4. Pick the best configuration of players for each configuration of water (choose_layouts.py)
   a. Since Dire Straits supports each team size for each map instance, each team is just a sequence of 4 starting locations. When the game is X vs X, only the first X players from each team will actually spawn. RMS conveniently ignores the remainder of the team.
5. Sort the chosen maps based on how balanced and “interesting” they are (choose_layouts.py & choose_layouts2.py)
   a. The overall goal is to pick the most balanced maps out of the millions that get generated.
   b. It’s non-trivial to algorithmically assign a number to how “interesting” an instance of a map is. Giving higher scores to instances where players aren’t just in a straight line or very close to each other (and doing the same for the straits) was an attempt at quantifying this.
6. Prepare the best maps for export, but skip the ones that are too similar to instances that were already chosen (choose_layouts2.py)
   a. The goal here is to make best use of the limited space by choosing a wide range of maps. Because the generation process is completely random, it doesn’t have any inherent duplicate protection.
      i. Assigning scores to maps and sorting them based on that would probably increase the number of duplicates at the top.
   b. However, the similarity-check isn’t very sophisticated. For example, two instances that are just reflections of each other or rotated versions of each other would not get detected as similar.
   c. This adds additional data which is feasible to calculate for a small number of map instances, but impractical to calculate for millions/billions of possible ones.
7. Generate the RMS (write_rms.py)
   a. Build the random selection logic that picks which map instance to use
      i. Avoid nesting if possible. RMS doesn’t handle it well.
      ii. Pay attention to the random chances because you have to use whole numbers. Generating random blocks that mix percent_chance 2 and percent_chance 1 statements for things that should have the same frequency results in very unequal frequencies.
   b. Combine this with the rest of the RMS code

It treats the river as a constant (because generating that uses some slow logic instead of just picking random points) and then looks at every single combination of lakes & player positions that would fit on it, using a compressed coordinate space. (Examining every possibility for every river in a 100x100 coordinate space would require an entire data center. This runs on a personal computer).

I rewrote the code to go through the pipeline breadth-first because I wanted to optimize each part of the process individually (for large quantities of map instances), allowing for the possibility of using a faster process (eg. not python) for some of the slower stages. I also thought that splitting things up would make it easier to reuse some parts and make it easier for others to understand.

I suggest going depth-first if you just want to make a RMS and not worry about that stuff. The RMS capacity ended up being lower than I expected, so making things run quickly is nowhere near as important if you only have space for a few hundred instances anyways. (You should still generate many more than that in order to choose the best ones.) Not trying to find player configurations that have to be balanced using the same sequence for 1v1 and 2v2 and 3v3 and 4v4 on an asymmetric map with intentionally random water is one way to increase the success rate.

Why look at every possible combination?

This type of brute-force random generation is inefficient and probably unsuitable for any real-time tasks (unless there's lots of precomputation/caching). However, the alternatives have a few disadvantages.

First, you wouldn't know when to stop looking. If you use something resembling a solver, tree search, or a greedy matching algorithm to do something like place players on a given map, how would you know when you've found a sufficiently interesting or balanced setup for that map? Some geographies have millions of possible player configurations. Some only have a handful. Stopping when you've found a fixed number of possibilities is premature in the first case, and not saving you any computation in the second case. Stopping once you've found 1% of the total possibilities isn't easy to do, and going much higher than that is probably slower than looking at all possibilities anyways. Stopping after a fixed amount of time has elapsed seems arbitrary. I don't have years of experience building truly random maps, so I couldn't resolve all these dilemmas.

Second, any procedure or intelligent algorithm is likely to reduce randomness. When the goal is to put players in new and unpredictable situations, a recipe that knows what it's looking for and finds it (eg. put players in a circle) isn't going to be the best way to accomplish that. This doesn't mean it's impossible to look through the possibility space more intelligently, but it's not easy to do so without introducing bias.

Finally, you don't know what you don't know. When I saw the output, it included player configurations that I hadn't even considered. The ability of random generation to surprise people is precisely why we use it. In some ways, I was quite strict with the balance for Dire Straits, and the code invalidates many of the more unusual layouts (probably prematurely). Figuring out how to guarantee balance without reducing variety is an open question and requires experimentation.

Exclude instead of Include

If you can generate maps which cover the full range of possibilities, then you can start with a large amount of randomness. As you exclude instances which fail to meet playability criteria, the map will become less random. This approach is mostly limited by how large the possibility space is, as iterating through it takes a lot of time and/or performance.

On the other hand, if you tell a program what to build, it would start without randomness. As you include more components, the map might become more random (depending on how random each of those components are). The resulting randomness relies on your ability to include it, which is probably difficult to design in addition to difficult to program.

In general, achieving performance via tools like approximation and parallelism is easier than programming ingenuity and creation that must account for all possibilities. For randomness specifically, the exclusion criteria and inclusion criteria should affect the resulting configurations in an unbiased way. If you have to do something like use a 10x10 grid instead of a 100x100 grid for performance reasons, the stuff you actually care about (how varied the map is) shouldn't get skewed in a certain design direction too much. However, if you have to come up with a complete set of features to add to the map generation, then the result would get skewed by the features you forget about or the ones that aren't random enough or the ones that aren't implemented perfectly.

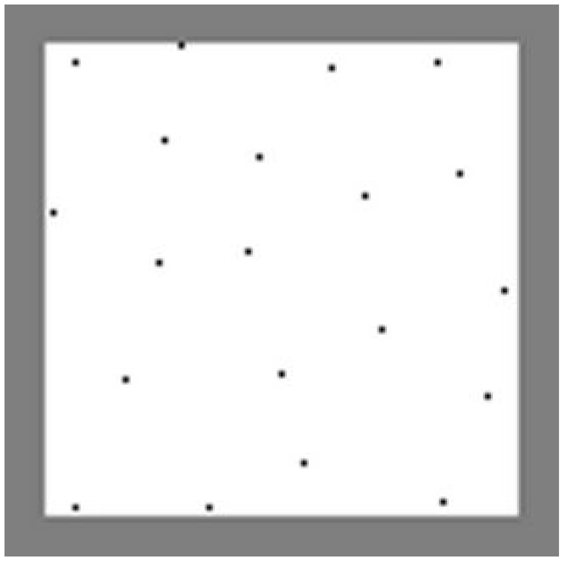
For Dire Straits, I used as much randomness as practical for the lake positions and player positions. However, the river was generated additively via a process with only some randomness. The purpose of generating the river was to create the specific geographic feature of straits, whereas the purpose of placing lakes and players was for the gameplay to be varied. I wish I had made the river generation process even more varied and random.

Some Implementation Details

Precomputed Random Patterns

A common task when making maps is to find all (or many) sets of coordinates that are separated by at least a certain amount (or at most a certain amount, or exactly a certain amount). For example, you may want to find starting locations for 8 players. If some of the possible locations are invalid (because you've already placed restrictive objects there or nearby), then the number of possible sets is going to be dynamic.

I think this is a slow process no matter how it's implemented. If you compute it on the fly as needed, it might not be ideal. I used precomputed patterns instead. For example, a single pattern with a border of 8 and minimum separation of 15 (measured with chessboard distance) approximately looks like this in a 101x101 coordinate space:



Many of these (millions) can be intersected with the map. If you have a fast way of computing that overlap, then it could perform better than real-time computation which would yield an equivalent amount of results.

Another benefit is that it keeps the runtime constant. If some maps have billions of possibilities and others only have hundreds, then looping through a finite set of patterns for both is a way to guarantee that the program isn't doing too much work for the former and is doing enough work for the latter. Constant runtime isn't necessary for this application (since it's an offline process), but it helps during development of something new when you're unsure about which parameters to pick and what the resulting workload might be.

Compressed Coordinate Space

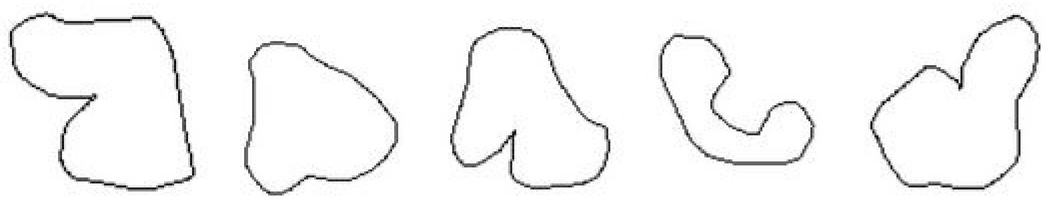
Examining the entire possibility space is difficult if the map's size is 101x101. For Dire Straits, the map (excluding some borders along the edge) is divided into 64 regions (with each region being 11x11), and lakes and town centers are only allowed to spawn at the center of each region. This allowed for the possibility to do some of the set intersection in a lower level language using a bitwise AND. I ended up doing all those bitwise operations in Python anyways, but I don't think the interpreter is actually using 64-bit integers behind the scenes.

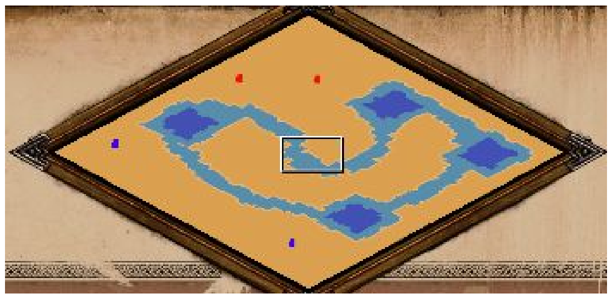
The downside of this is that it fails to find maps for rivers where the region centers alone aren't a sufficient part of the solution space. Usually it finds some "valid" configurations, but not a large enough quantity of them to pass all the balance checks. A beneficial side-effect is that it makes cheating more difficult since using a program to read the exact locations of you and your teammates doesn't narrow down the list of possible map instances too much. It's possible to do some randomization of these locations instead of always spawning things at the center, but it depends how accurate you want the balance checks to be.

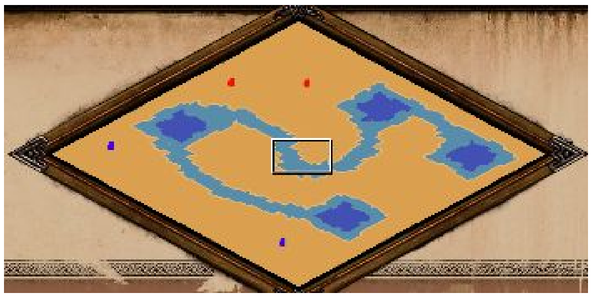
Things that didn’t go well

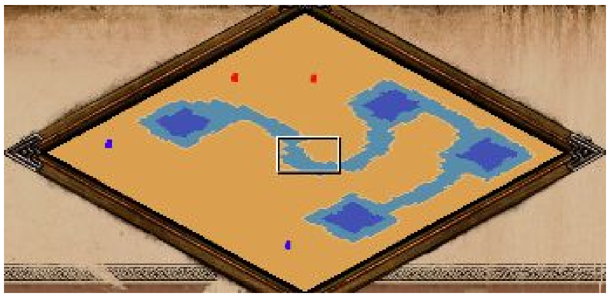
Increasing Randomness with Looped Straits

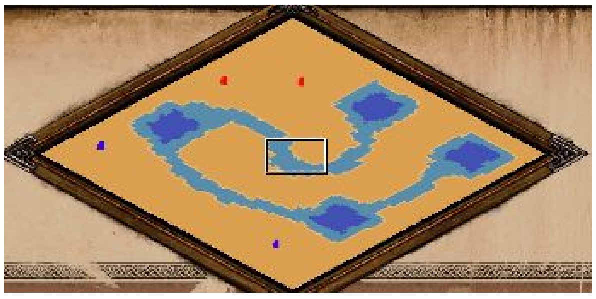
I knew that RMS capacity was somewhat limited, so the idea behind this was to generate rivers that could all be further randomized by a single percent_random block in the RMS. The code refers to these as “loops.”. For rivers with this property, Instead of generating the same straits every time, the game may hide one of them at random. This would sort of increase the number of maps without increasing the size of the script too much. However, after all the sorting and filtering, only a single “loop” made the final cut. I did not intend for it to be so rare, but I didn’t intervene to change that. Whatever the algorithms decided was best was what I went with.

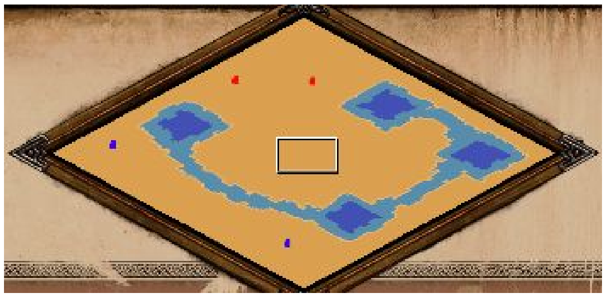












I suspect that the reason there’s only one of these in the original release version of Dire Straits (it’s not the one shown here) is that the code to generate the possible valid player positions doesn’t actually look at every single map size, so with the filtering criteria it’s hard to fit 8 players on loops when water covers so much of the map. Prior versions of the code that operated on the full coordinate space (but not the full possibility space) the whole time did find more solutions for these “loops,” so they can be playable. However, they probably don’t score as highly as other maps. Given that these don’t actually filter to the top, calculate_routes would run much more quickly if it wasn’t tasked to find these “loops.”

The Possibility Space is Underexplored for Smaller Sizes

Ensuring that each instance that gets exported works for every single team size is good for saving RMS space, but it has some consequences. Every 1v1 map is a 4v4 map that doesn't generate 6 of the players. The balance checks are still accurate since they check each map 4 times (for 1v1, 2v2, 3v3, and 4v4), but maps which don't work well for 4v4 are probably underrepresented if the goal is to find the largest variety of maps for a smaller size.

This could be addressed by allowing for more asymmetry for how big & safe from water player bases are within a team. The land is a much larger percentage of the map on larger map sizes because the size of lakes and width of straits doesn't scale with map size (which is intentional). On larger maps, things tend to be much further away from water (in terms of number of tiles). Instead of giving every player a huge base, it's possible to place some players much closer to the water, as long as the balance between two teams is the same. The code supports this kind of logic (see get_info_grid_u), but it could go a lot farther than it does. Doing so would probably require more complicated balance checks.

Precalculating Route Information

I don't remember exactly what happened, but I wanted to optimize the operations in calculate_layouts.py. This led to caching some data with precalculate_route_info.py. I'm sure it sped up calculate_layouts a lot, but it seems like many of the precalculation operations should just be performed in calculate_layouts instead.

Looking Back and Looking Forward

As non-random maps keep taking up a larger share of the RM map pool, it feels like one of the defining features of Age of Empires is disappearing. When that's combined with some new features like perfect circle placement and team positions and matchmaking that makes it foolish to pick random civilizations and impractical to play on different maps, the aggregate effect is that games become much more deterministic and predictable than what the design was ever about. Exploration is being replaced by memorization. Adaptation is being replaced by preparation and build orders. Strategy is being replaced by tactics.

If you listen to Ensemble developers talk about the game and their vision, the map design was much closer to games like Civilization than to games like Warcraft & Starcraft. If you install the original AoE2, the default map is coastal, not something hyper-focused on linear play patterns. Contests of execution are fine if they're just a part of the game and not the entire game, but I'll caution that only focusing on what supposedly makes for good viewing just recreates the conditions that led to the decline of these kinds of RTS games in the first place. Maybe multiplayer RM needs to be split into Fixed Map and Random Map if the competing interests of e-sports and fundamental Age of Empires design aren't reconcilable. That stuff is way beyond my control, and I mostly play turn-based games now anyways. But I hope I've shed some light on one way to put the 'random' back in 'Random Map' for anyone who desires to do so.

Contact Info:
bizs.email.address@gmail.com