Wordcrossing Level Generation

Harry Keightley

June 22, 2025

1 Background

Wordcrossing is a little game I made in a weekend, just after wordle blew up in 2023. It's a little blend of crosswords/scrabble and wordle, where the game is essentially:

> Given a start and goal position on a 2D grid, and a set of letters, construct a series of valid english words that connects the start and goal positions.

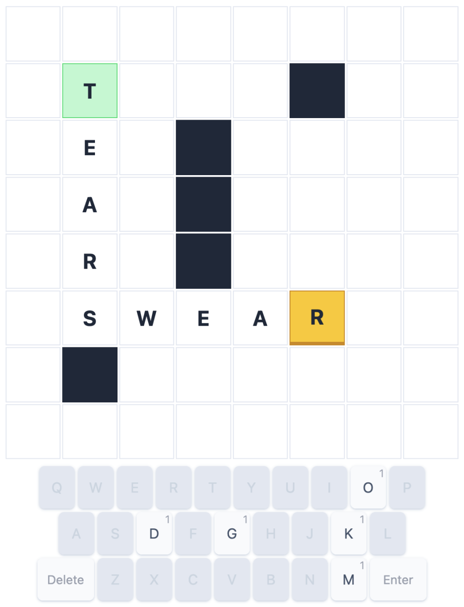
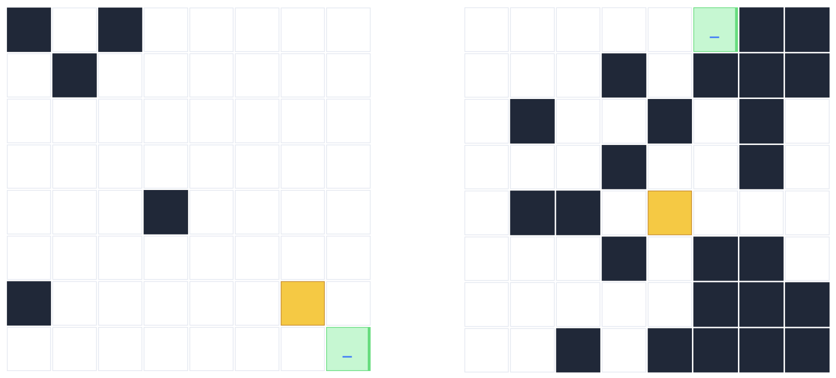


Figure 1: Example of a solved wordcrossing

In figure 1: the start position is the green square; the goal position is the yellow square; and the letters at the bottom of the screen + "tearswear" form the total letters that can be used to form solutions.



(a) Too easy (b) Too hard

Figure 2: Showing differences in level difficulty

In the online version of wordcrossing, players are incentivised to find a solution that maximises the total letters, and minimises the number of words placed. This leads to a situation where players (being myself, my partner, and my mother), will iterate on their solution a few times to compete with eachother.

If the daily levels are too hard, or even too easy (figure 2), it has a big impact on player fun— turning the level into a slog or a let down, respectively. Suffice it to say, finding that right balance of difficulty is an important problem to solve for this game, and a surprisingly fun little task.

I tend to use wordcrossing.com as a staging area to test new technologies I'm interested in, so I have previously written this level generation library in:

- Python, when the project was first created, running in a container on Google Cloud, invoked on a cron job, where it would write the output to a bucket.
- Typescript, when I created a monorepo for generation and playing the game, and to reuse the types between frontend and backend. In this iteration, I used cloudflares functions and workflows for the same functionality as GCP.
- And now, Rust, to speed up the level generation process. Depending on our criteria in generating levels, small tweaks can make large differences in runtime, and I was frequently getting time-outs in the typescript implementation. Using rust is also a great excuse to use rust.

2 Definitions

Before we jump into the level generation process, it would be good to have some common ground for terminology.

Position

A tuple of integers, (row, column).

Neighbours

The neighbours of a position are those directly up, down, left and right of it.

Entity (Wall)

In the level generation part of wordcrossing, the only relevant entity is the `Wall` entity, which is impassable, meaning you cannot place letters atop it.

Grid

A 2D grid of a given rows and columns. Contains entities.

Free space

The free space in the Grid is the set of positions that aren't Wall entities.

Rooms

A room, R in the grid is the maximal set of positions such that for any two positions, $a, b \in R$, there exists a path in the free space, $[p_0, p_1 ..., p_n]$ where:

- $a = p_0$
- $b = p_n$
- For $i \in [1..n]$, p_i neighbours p_{i-1}.

Maximal here means no more positions can be added to R and have the above hold, i.e. a room is entirely bordered by walls, or the edge of the bounding grid.

Level

A Level is a struct of:

- A grid.
- A start and goal position on the grid.
- A HashMap of letters to their counts that could be used to construct words.

A level is valid iff:

- There exists a room in the grid such that the start and goal are in the room.
- The set of letters can be used to create a valid solution from the start to the goal.

3 Level Generation

A very informal summary of my level generation solution is as follows:

Setup:

1. Initialise an empty Grid of the given dimensions.
2. Setting up some initial walls.
3. Find all rooms in the grid.
4. Wall off every room except the biggest one.
5. Generate a distance map and turn map of the level.
6. Choose a start and goal position in this room, based on some metrics.
7. Create a level with an empty letter count.
8. Iteratively build up words from the start to the goal, based on the shortest path to the goal.

Let's dig into some of those in more depth.

3.1 Initial Walls

We randomly choose some percentage (between 15% and 50%) of positions in the grid to wall off. This gives a bit more variety to the level.

3.2 Exploration

Exploration— building rooms within the Grid, is done by calling DFS on each position in the free space (if we haven't already seen it during one of the last rooms DFS.

3.3 Final Walling

The initial walling both looks jagged, and makes it less clear to the player which positions they can validly reach from the start and goal. Walling off the unreachable positions in the level fix this.

To do so, we simply choose every room in the grid except the largest, and fill it with walls.

3.4 Distance and Turn Mapping

For future metrics, it's important to know how far every square is away from eachother (governing word length), and how many turns are required to get there (governing the number of words in the solution).

We generate distance and turn maps through Djikstras algorithm.

3.4.1 Distance Map

For the distance map, this is Djikstras(note: maybe not?), but calculating minimum distance between any two points in the graph.

In this version, the free space positions are nodes in the graph, with edges only existing between neighbours, and the weight of each edge being 1.

Basically, we initialise an edge map, $E = (position \to position \to distance)$ with $E(v, v) = 0$ for each vertex v in the graph.

Then we continue to iterate the graph until it converges, asking:

Algorithm 1 Finding smallest distances between all positions

```
changed ← $true$
while changed do
    changed ← $false$
    for all position do
        for neighbour of position do
            for destination in keys(dist(neighbour)) do
                $d_n \leftarrow 1 + dist(neighbour, destination)$
                if $d_n < dist(position, destination)$ then
                    $dist(position, destination) \leftarrow d_n$
                    $changed \leftarrow true$
                end if
            end for
        end for
    end for
end while
```

There are a few ways to improve the algorithm itself, though who doesn't love 4 separate tiers of nested loops? One idea (alg 2 to use a queue to store nodes who have the potential to change, instead of just iterating through all nodes each loop. The idea there is that we only have to recalculate the distance map for a position when one of its neighbours have new information.

In hindsight (and too late to change for this meetup), we also don't really need this distance information between any two points, as this was a hangover from an earlier iteration of the level generation algorithm. Instead we just need the

Algorithm 2 Finding smallest distances between all positions v2

```
changed ← $Queue < Position > ()$
queued ← $Set < Position > ()$
dist ← $EdgeMap < int > ()$

for all position in FreeSpace do
    changed.enqueue(position)
    queued.add(position)
    dist(position, position) ← 0
end for

while !changed.isEmpty() do
    position ← changed.dequeue()
    queued.remove(position)
    for neighbour of position do
        for destination in keys(dist(neighbour)) do
            $d_n \leftarrow 1 + dist(neighbour, destination)$
            if $d_n < dist(position, destination)$ then
                $dist(position, destination) \leftarrow d_n$
                if !queued.contains(neighbour) then
                    queued.add(neighbour)
                    changed.enqueue(neighbour)
                end if
            end if
        end for
    end for
end while
```

distances from the start (after we randomly choose it) to every other position. Oh well.

A more interesting idea, probably just because I've never encountered it before, is using roughly the same algorithm to make a turn map. In this algorithm, instead of storing distance to the goal, we store:

1. The number of turns to the goal
2. The next direction you would need to head in to get there.

The algorithm changes as follows:

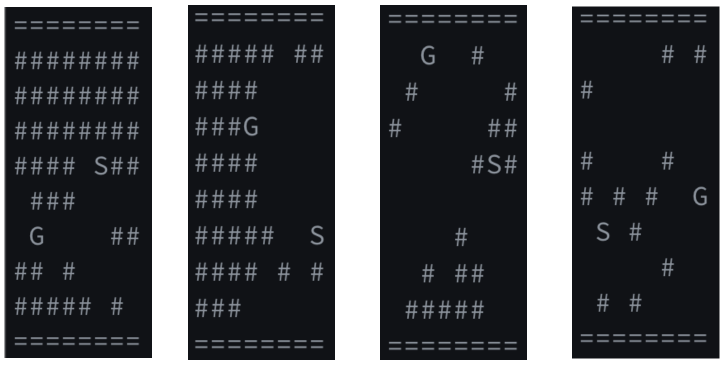
Algorithm 3 Finding smallest turns between each position

```
changed ← $Queue < Position > ()$
queued ← $Set < Position > ()$
turns ← $EdgeMap < (int, Option < Direction >) > ()$                ▷ (1)

for all position in FreeSpace do
    changed.enqueue(position)
    queued.add(position)
    turns(position, position) ← $(0, None)$
end for

while !changed.isEmpty() do
    position ← changed.dequeue()
    queued.remove(position)
    for neighbour of position do
        for destination in keys(turns(neighbour)) do
            $(turns_n, direction_n) \leftarrow turns(neighbour, destination)$
            if position.direction_to(neighbour) $!= direction_n$ then    ▷ (2)
                $turns_n \leftarrow turns_n + 1$
            end if
            if $turns_n < turns(position, destination)[0]$ then
                $updated \leftarrow (turns_n, position.direction\_to(neighbour))$
                $turns(position, destination) \leftarrow updated$
                if !queued.contains(neighbour) then
                    queued.add(neighbour)
                    changed.enqueue(neighbour)
                end if
            end if
        end for
    end for
end while
```

Notes:



```
========
########
########
########
#### S##
 ###
 G    ##
## #
##### #
========
```

```
========
##### ##
####
###G
####
####
#####  S
#### # #
###
========
```

```
========
  G  #
 #     #
#     ##
     #S#

    #
  # ##
 #####
========
```

```
========
     # #
#

#    #
# # #  G
 S #
     #
 # #
========
```

Figure 3: Start and goal choices

1. The direction from a position to itself is none in this case.
2. We only increase the turns if the direction from the neighbour to their destination is not the direction from the position to its neighbour.

3.5 Choosing a Start and Goal

Now with our distance and turn map, we have everything we need (and more), to choose the start and goal positions. Since we've walled off every room except the largest, we know all positions in the freespace are reachable from eachother. We can then pick any two positions in this space for the start and goal positions.

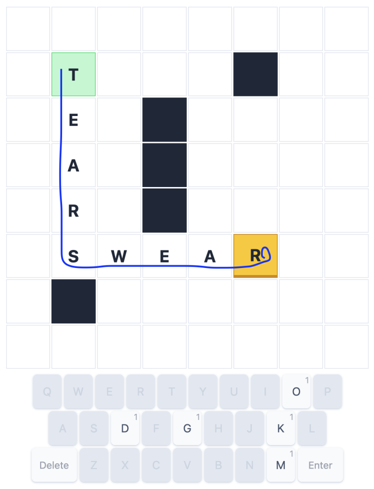
1. We start by randomly selecting a start position from the free space.
2. In selecting a goal, to make sure we don't pick positions that are too close, or with too many turns, we first sort all candidates ($FreeSpace - start$) in descending order of the sum of:
   - Their distance from the start.
   - And the thir number of turns from the start.
3. We then select a goal randomly from the first third of the sorted candidates.

Figure 3 includes a few generated maps, to get a feel for this. Note that the furthest distances from the start are not always interesting.

Now that we have our map, and our start and goal positions, it's time to try and find a solution!

3.6 Solving for a Level

To find a solution for our new level, we start by tracing a minimum turn path from the start to the goal node, with the help of our turn map. Note that the


(a) The minimum turns solution

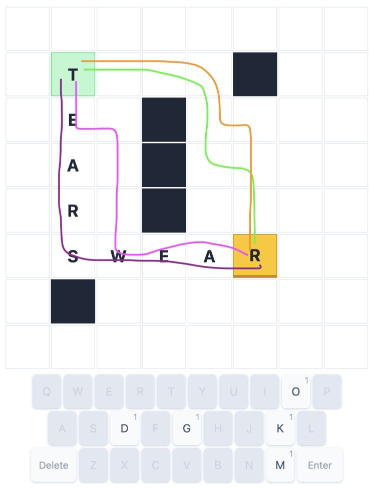

(b) Some min-distance solutions

Figure 4: Comparing solutions

minimum turns path is always the minimum distance path to the goal, with the set of minimum-turn paths between start and goal being a subset of the minimum distance paths (See figure 4).

Next, we convert this path to a series of junctions $(j_0, j_1, ..., j_n)$, between which we will place our words (figure 5).

We can then transform our problem of finding a solution, to successively finding words in our wordlist that satisfy a set of constraints. When we have no words in our solution so far, we attempt to place a word between the junctions (positions), j_0 and j_1, with the only constraint being that the word must have length: $dist(j_0, j_1) + 1$.

For the next word, we have a similar constraint: $length = dist(j_1, j_2) + 1$, but we additionally have the constraint that the first letter of this word must be the last letter of the previous word. I've called this an "index" constraint in the code. This index constraint doesn't always line up the way I've just described. In wordcrossing, a valid english word can only be placed from left to right, and from top to bottom. This means that sometimes, the previous word puts an index constraint on the *last* letter in the new word (imagine switching the start and goal positions in figure 5).

We continue to place words until we've either created a solution that connects the start and goal, or we've determined no solution exists given the previous used words.

There are a number of technical decisions here to address, here which I will cover in the next sections.

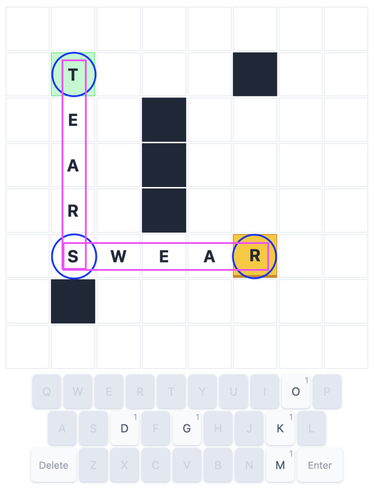


Figure 5: Finding junctions (blue circles), and placing words (purple rects).

3.6.1 Wordlist choice

The choice of the wordlist here can greatly affect how easily players reach a solution. I use an "easy" wordlist for level generation, and a more permissive wordlist when you're actually playing the game.

If we choose a permissive wordlist for level generation, it can come up with solutions containing words that no (modern) human would understand, which can make the day's letters perplexing when you're given an odd combination of "x, z, q, i, and w".

The wordlists for generation and playing the game are supplied in the `assets` folder for this repository.

3.6.2 Wordlist Data Structure

We want to make word lookups efficient for:

- Length lookups
- First letter lookups
- Last letter lookups

One potential choice here would be to have two Maps of:

$$WordLength \rightarrow StartLetter \rightarrow Set < Word >$$

Where the second Map stores each word in the set as reversed, so you can quickly lookup the last letter.

I chose the much quicker to implement data structure of a single map:

$$WordLength \to Set < Word >$$

3.6.3 Choosing words to fit the constraints

Regardless of the used data structure for the wordlist, you'll want some function which takes a series of Constraints as inputs, and gives a randomly selected word that fits the constraints, if one exists. The randomness is important in this case, or you may artifically limit your solutions.

A much cooler idea you could employ here to avoid missing potential solutions, is to prioritise words that make the subsequent constraints *easier*. The key idea here, is that some *Index Constraints* are more permissive than others— finding valid english words that end in I, is much harder than finding those that end in a consonant like S, D, T. Oddly, when you actually play wordcrossing, this is one of the techniques first you learn with experience to make your life easier.

So on that previous idea, you could create a distribution of start/end letters for a given word length, and then sample this to prioritise certain words in the solution.

3.6.4 Handling Failure

When we determine that no solution exists for a constraint, this doesn't mean that no solution exists for the level as whole, and there are a number of ways of solving this. In order of complexity:

1. `Scorned Lover`- Generate an entirely new Level and solution. This is what I used for my generator, with the intuition that if a solution fails to generate once, it's more likely that it'll fail twice, i.e. the level may just be harder to generate solutions for and not worth our time.

2. `Backtracking`- Reselect the previous word, in the hopes that a solvable constraint exists afterwards. A fancy backtracking solution could determine if it was possible to build a wordcrossing solution along the minimum-turn-path or not.

3. `Get fancy`- Don't restrict your solutions to minimum-turn-paths. I have not explored this, but this seems like a massive explosion in complexity for the programmer, for little benefit.

When we've come up with a solution, we attach it to the level to be inspected in the post-generation steps.

4 Post Generation

We have a level! But we're not done yet.

4.1 Evaluating Level Quality

How do we know if our solution is any good?

For my purposes, I discard any levels for which their solution's words have an average letter count of < 4. This is a huge discriminator on an 8x8 grid, but there's a massive increase in player creativity in the jump from 3-4 letter words, so I thought this important to prioritise. I may in the future reduce this back to 3, or something in between.

4.2 Player Creativity

Our solution has hinged around the strategy of finding a minimum turn path from start to goal. But what if our chosen words are arcane? What if the player want to take another path? To differentiate a good and bad wordcrossing player, and hopefully safeguard against any odd words in the solution, we can give them a few more letters to play with. This opens up the potential for interesting paths to the goal, and ways for the player to get much better scores than if they had just built directly to the goal.

To do this, we find the letter frequency in words of the wordlist, and randomly sample $len(solution)/2$ extra letters to pad out the solution.

5 Summary

So as a result of all this, we know that for any daily wordcrossing,

1. There exists a solution of somewhat easy words from the start to the goal positions, along the minimum turn path.
2. There likely exists multiple solutions and variation in potential paths due to the extra padded letters.

This is important to know when playing, as I could say to friends early on, ”no, no, I can promise you it's solvable.”

So far, we've never encountered a day that we couldn't solve manually. I used to scramble the solution in the JSON presented to the client so they wouldn't snoop, but I'm unsure anybody would have looked at that anyway.