Graphbased Wavefunction Collapse - Technical Documentation

Disclaimer: This overview primarily explains the reasoning behind various implementation decisions. Additionally, I share some of my visions for the project's future development. This documentation is intended to be read alongside the code itself, as the most important methods are accompanied by their own comments inside the source files.

- Graph representation
- Rulebook
- Solver
- Wrappers
- Dependencies
- References
- Acknowledgements

Graph representation

The graph representation used (named `WFCGraph`) is a List of `Node`s, the graph keeps and updates its number of total nodes set (node "is set" when any non-negative number has been assigned to it). Every `Node` has two Lists filled with other nodes: Parents and Children, these are a subset of other nodes in the graph.

This representation ensures quick access to the parents and children of a given node, eliminating the need to search through a list or set of edges.

Every node has its own ID that is assigned to it when the graph is being constructed. When given a graph with N nodes, the IDs range from 0 to $N - 1$. Indexing with M to `WFCGraph.AllNodes` (all nodes are stored here) returns a node with ID equal to M.

Graph constructors create new `Node`s and then build relationships between them.

Edges

`Edge` is a struct with two fields: Parent and Child. Meaning that, if the graph is directed, the edge goes from Parent to Child.

Loading graph from a file

A graph can be loaded from a file using the static method `WFCGraph.ParseEdgesFromFile`. Every edge has to be on its own line written as two numbers with a single space between them: `A B` this will be interpreted as `Edge(Parent: A, Child: B)`.

Edges do not have to be in any particular order. When representing an undirected graph, use only a single line to represent an undirected edge: "`A B`", the file should **not** contain an edge "`B A`". Note that a graph with N nodes has to be represented by edges with `A` and `B` between $0, N - 1$.

Example

```
0 1
0 2
0 4
1 2
...
```

Graph directedness

Information about graph's directedness is used in its constructors. User passes a list of `Edge`s and they are then interpreted either as directed or as undirected. When edge `(A,B)` interpreted is directed the algorithm adds `A` as a parent of `B` and `B` as a child of `A`, when interpreted as undirected the same is repeated but with `(B,A)`.

GraphConverter

A basic converter for graphs from the `QuikGraph` library to `WFCGraph` is provided. The method returns a new `WFCGraph` and also a mapping of `QuikGraph`'s Vertices to `WFCGraph`'s Nodes.

Keep in mind that this method is as general as it can get, it does not use any tricks for specific graph representation provided by `QuikGraph`. It takes some (precious) time to convert between the two graphs. Constructing `WFCGraph` from a list of edges is the preferred way to go.

Rulebook

Rulebook is a collection of rules that, when asked by a type returns a single `Rule`. It has two arrays of rules, one for the relationships between a parent and children (that is passed to a constructor by the user) and one for the relationships between a child and parents (that is created automatically during initialization).

This way, when assigning a value M to a node K, we can easily retrieve rule for children using `rulesForChildren[`M`]` and rule for parents using `rulesForParents[`M`]`, and update options of K's neighbors accordingly.

Rule

`Rule` is a description of a relationship between a parent node and its children represented by a struct with two fields: Item and Options. Item is the value assigned to parent and Options are all allowed values for its children.

In the set of rules in 1, there are for types of nodes `A`, `B`, `C` and `D`. `A` can have any type as its children and `B`, `C`, `D` can have only `A` as their parent. Note that by default rules are oriented (parents on the left, children on the right).
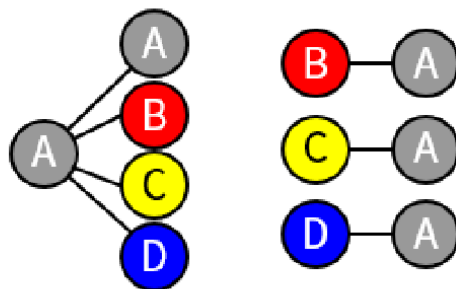

Figure 1: Example of rules, https://ieeexplore.ieee.org/document/8848019

Loading rules from a file

The rules from above can be represented as a JSON and the parsed calling `RulebookParser.RulesFromJSON`, it uses the `System.Text.Json` library.

```
{
  "0": [ "0", "1", "2", "3" ],
  "1": [ "0" ],
  "2": [ "0" ],
  "3": [ "0" ]
}
```

Note if there are N types of nodes, then there must be rules with types from 0 to $N-1$.

Solver

The heart of this whole project is the Solver, a wave function collapse implementation for general graphs.

The Solver is initialized with a `Rulebook`, optional `globalWeights` and `evaluateNode`.

`globalWeights` is an array of type frequencies, the higher the number at index M, the higher to probability that type M will randomly chosen (when assigning values to nodes). If no array is given, the frequencies are initialized as a list of ones (for example, in solved Sudoku all numbers have the same frequency).

`evaluateNode` is a method (`EvaluateNode` is a delegate, a template if you will) that, when given a node and `globalWeights`, calculates node's entropy. Users are able to define their own methods for node evaluation.

```
public delegate double EvaluateNode(IEnumerable<int> nodeOptions,
                                    IReadOnlyList<int> globalWeights);
```

Shannon Entropy - the default

The Shannon Entropy of a given node can be calculated as:

$$\mathrm{H}(X) := -\sum_{i=1}^{n} p(x_i) \log p(x_i)$$

where $p(x_i)$ is the probability of outcome x_i.

After few updates to node's options, some of the types may not be valid, so their probability is zero and all other probabilities have to be scaled accordingly. Method that does this is already implemented as `ConvertWeights`, it can be used in other user-defined methods.

Algorithm: ShannonEntropy

Input : Node.Options, GlobalWeights
Output: `double`

```
// get sum of frequencies of valid options
```
1 total ← 0
2 foreach *option in Node.Options* **do**
3   total ← total + GlobalWeights[option]
4 end

```
// normalize the probabilities
```
5 probabilities ← []
6 i ← 0
7 foreach *option in Node.Options* **do**
8   probabilities[i] ← GlobalWeights[option] / total
9   i ← i + 1
10 end

```
// the actual Shannon Entropy calculation
```
11 entropy ← 0
12 foreach *prob in probabilities* **do**
13   entropy ← entropy - (prob × log(prob))
14 end
15 return entropy

PrioritySet

For quick enqueueing and dequeueing of nodes to process the algorithm uses a `PrioritySet` implemented by eiriktsarpalis, here. The structure allows updates priority of nodes in $O(\log N)$ and does not allow duplicates.

Working with it is similar to C#'s basic PriorityQueue, the ability to update already enqueued elements is the only difference.

Constraints

Imagine you are given a Sudoku problem, you don't start with an empty grid, rather with some of the values filled in - let's call them constraints aka "the cells set before any solving takes place".

List of constraints can be passed to the Solver, there are two ways how to represent a constraint: by a reference to a node itself or by node ID (the latter gets converted to a list of references). For this purpose two structs are implemented `ConstraintByNode` and `ConstraintById` both with two fields: node specification and the ForcedValue.

When applying constraints, the Solver propagates these choices as if it was a normal value assignment. The calculation time is significantly shorter than if we would only assign the values without propagation.

Main algorithms and methods

Solve Method

Algorithm: Solve

Input   : Graph, Constraints, GlobalWeights
Output: Graph or `null`

  // initialize options for every Node in Graph
1 **foreach** *Node in Graph's nodes* **do**
2 | Node.Options $\leftarrow [0, 1, 2, ..., N-1]$
3 **end**

  // apply constraints
4 **foreach** *Constraint(Node, ForcedValue) in Constraints* **do**
5 | Node $\leftarrow$ ForcedValue
6 | **Propagate**(Node, ForcedValue)
7 **end**

  // initialize PrioritySet
8 **foreach** *Node in Graph's nodes* **do**
9 | PrioritySet $\leftarrow$ enqueue (Node, **EvaluateNode**(Node, GlobalWeights))
10 **end**
11 **return** **RecursiveSolve2**(Graph, PrioritySet)

RecursiveSolve2 Algorithm

Algorithm: RecursiveSolve2

Input   : Graph, PrioritySet
Output: Graph or `null`

1 collapsingNode $\leftarrow$ dequeue from PrioritySet
2 copy and store options for collapsingNode's children and parents
3 **while** *collapsingNode has some options* **do**
4 | value $\leftarrow$ randomly choose from collapsingNode.Options
5 | collapsingNode.Options.Remove(value)
6 | collapsingNode $\leftarrow$ value
7 | **Propagate**(collapsingNode, value)

8 | **if** *propagation is successful* **then**
9 | | **if** *Graph is solved* **then** // all nodes in Graph are set
10 | | | **return** Graph
11 | | result $\leftarrow$ **RecursiveSolve2**(Graph, PrioritySet)
12 | | **if** *result is not **null*** **then** // recursive assignment was successful
13 | | | **return** result
14 | **else**
15 | | reset child and parent options of collapsingNode
16 **end**
  // assigning any of the options left for collapsingNode does not lead to solving given Graph
17 PrioritySet $\leftarrow$ enqueue collapsingNode
18 **return** `null`

Wrappers

As mentioned before, the implemented algorithm is very versatile and can be used to perform many graph-based calculation. I have already implemented some wrappers for the Solver, that can be an inspiration for others. All of them implement the `ISolver` interface that is quite simple:

```
public interface ISolver<T>
{
    public T? Solve(T problem);
}
```

All of these usages have a working code example implemented in the project.

Graph Coloring

> **Note**: To use this example to its full extend you need to install the GraphViz library (graph drawing engines).

> And, technically, this is not a wrapper.

Rulebok has a method `CreateColoringRules(int n)` that returns a list of rules for a set of colors $\mathbf{C}$ such that: $\{(c, \mathbf{C} \setminus c) \mid \forall c \in \mathbf{C}\}$ (every color can be a adjacent to every color except itself). The base Solver is used to get the coloring and then methods `CreateImage` and `ShowImage` are used to render and show the result.

Uses a list of 1024 pregenerated contrastive colors to color the graph using the DOT format and GraphViz to generate final image.

When using the GraphViz engines, use `dot` for any graph where there is any type of hierarchy and `fdp` for anything else.

Grid Generation

> **Note**: I will not be implementing the option to use and apply rules to larger sub-grids of pixels (as done by mxgmn) in the nearest feature for two main reasons:
>
> 1. It is significantly easier to implement this with an algorithm that operates on grids by default.
> 2. This feature has been implemented multiple times before and is not the primary focus of this project. Instead, I prefer to concentrate my efforts on general graphs.

The wrapper for grids generates all the necessary edges which are then used to construct an undirected `WFCGraph`.

As before, the Rulebok has a method `CascadeRules(int n)` that returns a list of rules for a set of color such that color M can be adjacent to colors $M-1, M, M+1$. With this we can define a "country gradient": water, sand, grass, dirt and rocks; and end up with a really simle and stupid Minecraft-like terrain generator as seen in 3.

Sudoku

Solving Sudoku is just a more complex graph coloring problem. One cell has more neighbors (relations) than what it may look like from the grid itself.

The wrapper generates all the necessary edges (seen in 4) for all cells and also retrieves constraints from the problem given. This information is then used to create a `WFCGraph` that is passed to the base Solver with retrieved constraints.
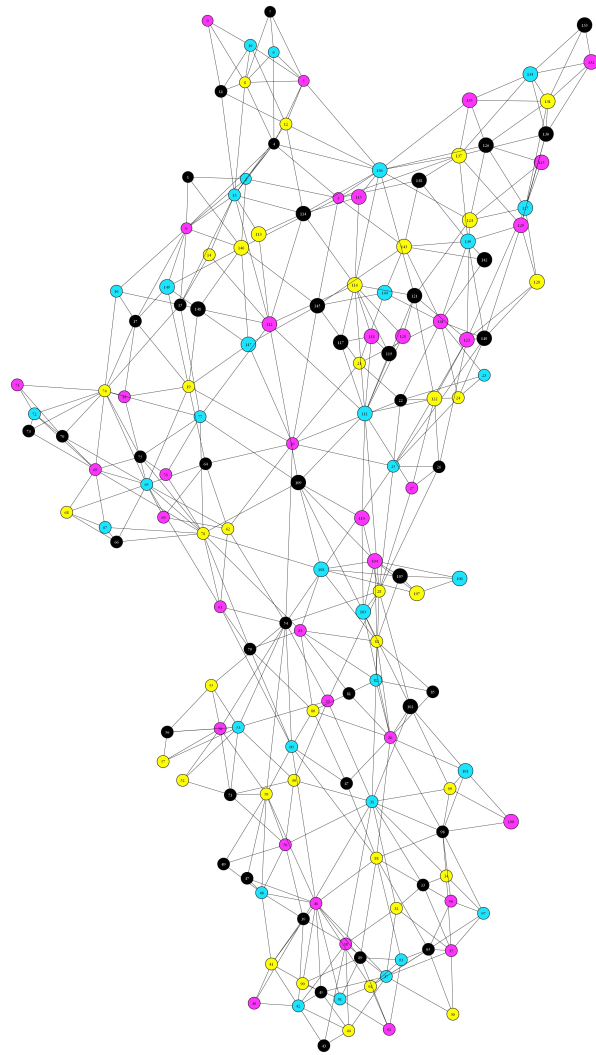
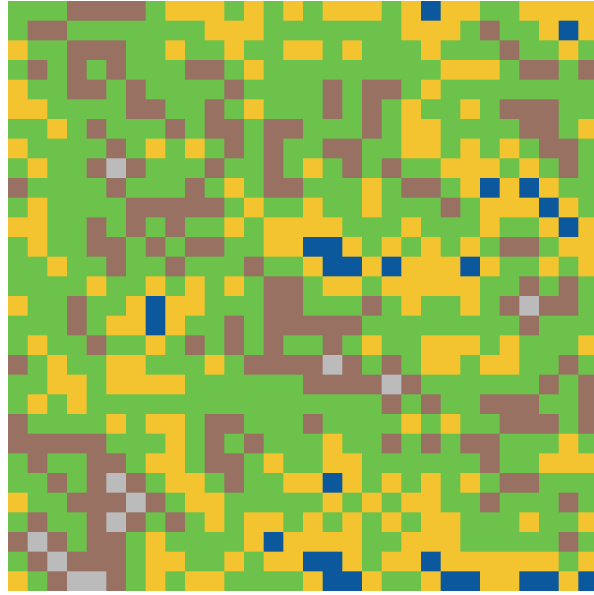Figure 2: Example coloring of a planar graph
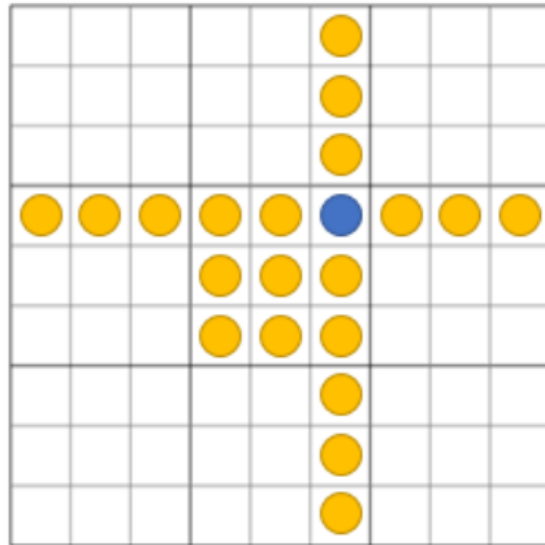
Figure 3: Primitive terrain generation

Figure 4: Neighbors (yellow) of a single cell (blue) in a Sudoku puzzle, https://ieeexplore.ieee.org/docu ment/8848019

The Sudoku board is represented as `int[9,9]`, for example:

```
{
    { 0, 0, 0, 7, 0, 5, 0, 0, 2, },
    { 0, 0, 3, 0, 0, 8, 0, 7, 0, },
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, },
    { 2, 0, 5, 0, 0, 0, 0, 0, 0, },
    { 0, 4, 6, 0, 0, 0, 0, 1, 5, },
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, },
    { 4, 0, 0, 5, 0, 0, 0, 8, 0, },
    { 6, 0, 1, 0, 8, 0, 0, 0, 0, },
    { 8, 7, 9, 6, 2, 0, 0, 5, 0, },
}
```

Map Generation

When creating a roguelike game one needs to not only generates the levels themselves but also the map which includes the levels. The rules may be strict and in the worse case the graph may include cycles.

Inspired by the game Peglin, I created a simple map generator and came up with some rules and frequencies. The map grows from one node into maximal width and then in collapses back into one node at the end (maps in the game can be much more complicated, but after all, this project is about coloring these maps, not creating them :)).

Figure 5: Peglin map example, https://store.steampowered.com/app/1296610/Peglin/

There are four types of levels: normal fight (black), boss fight (yellow), shop (blue), treasure (purple); with frequencies `[10, 5, 2, 2]`. The player can only progress downwards, so the graph is oriented. Also we can use pass a constraint to the Solver so that the player starts in a normal fight. The rules are same as in 1:

```
{
  "0": [ "0", "1", "2", "3" ],  // normal fight
  "1": [ "0" ],                 // boss fight
  "2": [ "0" ],                 // shop
  "3": [ "0" ]                  // treasure
}
```
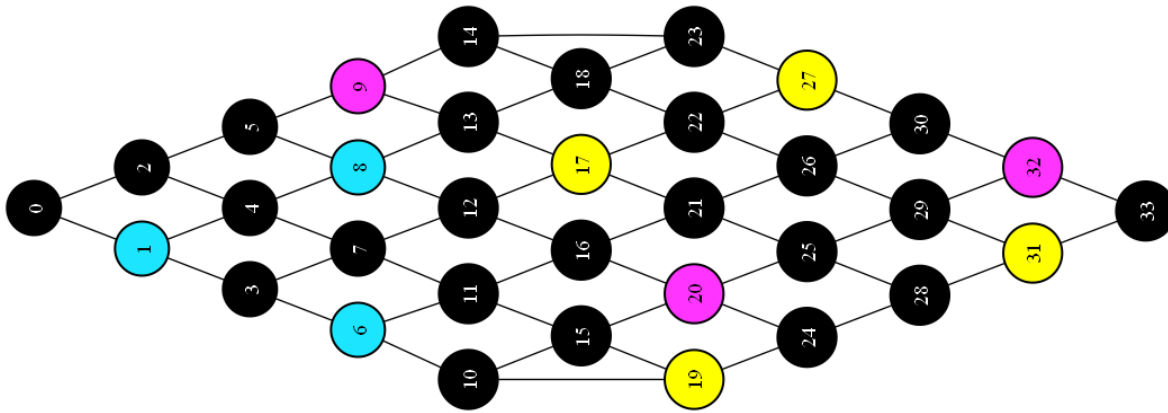
Figure 6: One of the possible results, the player would progress from left to right

Dependencies

- DotNetGraph
- QuickGraph
- Refit
- System.Drawing.Common

References

1) Automatic Generation of Game Content using a Graph-based Wave Function Collapse Algorithm, by Hwanhee Kim, Seongtaek Lee, Hyundong Lee, Teasung Hahn, Shinjin Kang

2) The Wavefunction Collapse Algorithm explained very clearly, by RObert Heaton

Acknowledgements