Smart Level Generation for the Routing Problem

(Industrial project with InstaDeep)
— Final Report —

Randy Brown, Ole Jorgensen, Danila Kurganov, Ugo Okoroafor, Marta Wolinska
{randy.brown21, ole.jorgensen22, danila.kurganov19, ugochukwu.okoroafor22,marta.wolinska16}
@imperial.ac.uk

Supervisor: Dr Robert Craven
Industrial Supervisor: Clément Bonnet, InstaDeep

Module: COMP70048, Imperial College London

May 2nd, 2023

1 Introduction

Jumanji [4] is an open-source library created and maintained by InstaDeep providing a diverse suit of environments for training reinforcement learning (RL) agents. One such environment undertakes the routing problem[1], where the aim is to connect heads and targets via routes in a grid. This can be visualised as the popular game Number Link [14], but other applications could include routing printed circuit boards (PCBs), logistics or manufacturing.

To successfully train RL agents one must set a solvable, non-trivial problem. If the solution is either too easy or impossible to find, the agent may not be able to learn useful policies. As such for the routing problem, boards which can be fully connected with sufficient difficulty are required, which is the focus of this project.

1.1 Project Specification

The goal of the project is to implement a generator for the routing problem, which produces complex but solvable boards. The existing baseline implementation in Jumanji contains a generator which randomly selects pairs of points within a grid, `UniformRandomGenerator`, however this does not ensure the setting of a solvable problem, nor does it allow gauging the complexity of generated boards.

Beyond the implementation of a suitable generation solution, an additional stretch goal was the conversion of the generator to use JAX[2] [5], in order to allow an open-source contribution into Jumanji.

2 Problem Definition and Approach

2.1 Problem Definition

As mentioned above the key features of the boards generated are solvability and complexity. A board is represented numerically as a (typically square) grid of non-negative integers. The numbers in each cell represent the type of component occupying the cell and only one component can occupy a cell at a time. The numbers are encoded, by the Jumanji environment, as follows: empty spaces (0), wire paths ($x \in \{x \equiv 1 \pmod 3\}$), heads ($x \in \{x \equiv 2 \pmod 3\}$) and targets ($x \in \{x \equiv 0 \pmod 3 \ | x > 0\}$), a numerical example is provided in figure 1(a).

For the purposes of the project the following definitions were set out. A wire (with head encoding x) is considered *connected* if there exists a single unbroken path (encoding of value $x - 1$) from that the head to the target (of value $x + 1$). Additionally, a wire with a head and target next to each other is also considered connected.

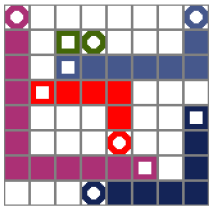
A board with n pairs of heads and targets is considered *solvable* if there exist n non-overlapping routes that connect each head to its target. A board is considered *solved* if all the wires on the board are connected with no floating heads, targets or paths with no intersections between wires, as shown in figure 1(b). Note that the solution to a board is not necessarily unique as there can exist multiple ways to connect heads and targets. A *training board*, example in figure 1(c) is a solvable board with all the wire paths removed. So it only contains heads, targets and empty cells. In addition to being solvable, a board should be non-trivial to solve. The board in Figure 1(d) is non trivial but is not solvable as the blue and purple wire's paths mutually exclude one another. The board in Figure 1(e) has little complexity and is trivially solvable.

[1] Routing is an NP problem that consists of connecting routes to their target on a grid/board.

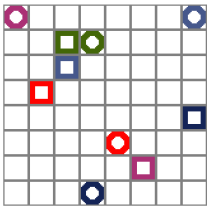
[2] JAX is an open-source Python library developed by Google that can improve computation time due to using Just in Time (JIT) compilation

$$\begin{pmatrix} 6 & 0 & 0 & 0 & 0 & 0 & 0 & 15 \\ 4 & 0 & 8 & 9 & 0 & 0 & 0 & 13 \\ 4 & 0 & 14 & 13 & 13 & 13 & 13 & 13 \\ 4 & \mathbf{2} & \mathbf{1} & \mathbf{1} & \mathbf{1} & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 11 \\ 4 & 0 & 0 & 0 & \mathbf{3} & 0 & 0 & 10 \\ 4 & 4 & 4 & 4 & 4 & 5 & 0 & 10 \\ 0 & 0 & 0 & 12 & 10 & 10 & 10 & 10 \end{pmatrix}$$

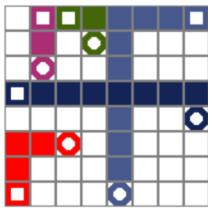
(a) Solved Board in Numerical Form



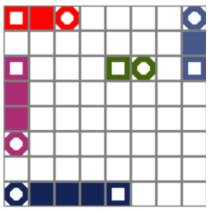
(b) Solved Board in visual form



(c) Training Board in visual form



(d) Non-Solvable Board



(e) A Solvable but Trivial Board

Figure 1: A solvable 8 by 8 board with 5 wires. In (a) the equivalent of the red wire is set in bold for easier translation between the figures.

2.2 Solution Design within an Existing Framework

The starting point of the project was the `Connector` environment within the Jumanji library (v0.2.0), which enabled quick experimentation set up and allowed the use of existing features such as board visualisation. As such, the solution to be designed needed to easily slot into the existing infrastructure and follow predefined behaviours.

The `Connector` environment contains multiple agents in a grid that each represent a start and end position that need to be connected in as few steps as possible. However, when an agent moves it leaves behind a path, which is impassable by all agents. Thus, agents need to cooperate in order to allow each other to connect to their own targets without overlapping. As the environment is intended for use in abstracting the routing of PCB boards, the terms wire and agent are used interchangeably.

It is important to note that the Jumanji library is fully written in JAX, which posed two key considerations for prototyping and delivery. First, in the research stage, where generators designed using NumPy needed to interface appropriately with the JAX code. Secondly, in the shipping stage, to enable the code to be used as a contribution into the library the final solutions selected needed to be converted to use JAX.

2.3 Solution Design - Prototyping

To enable fast prototyping and a coherent structure for development two design elements were crucial. An `AbstractBoard` class was established to pre-define the minimum required structure for each generator, thus allowing flexibility to each generator while ensuring they could be initialised and used in a coherent fashion. This was necessary to build an effective interface to work with any generator from anywhere in the code base without the need for specific handling.

The *interface*, based on the factory method [15], which used two classes: `BoardName` and `BoardGenerator`. `BoardName`, contained an Enum with all available board generators, whereas `BoardGenerator`, was used to match a `BoardName` with its class and return this where required in the code. This meant that at any stage, by only specifying the Enum, the desired board generator class could be returned, initialised and used.

2.4 Solution Design - Final Implementation

The final implementation, as expanded upon in section 4, was completed in JAX. This posed several challenges that will also be discussed in section 4.4. The delivered implementation consisted of two board generators: one based on the random walk and one which adopted ideas from more advanced generators. The former had to meet the contribution guidelines as set out by Jumanji and be fully unit tested to ensure it was production ready. The latter could be delivered as part of the research repository, where with a few improvements it could be easily made production-ready. Although the more advanced generator was expected to supersede the simpler random walk generator, the latter was delivered first in order to minimise any risks to do with the delivery of the more complex generator in JAX. This ensured that at minimum a solvable board generator could be merged into Jumanji.

2.5 Team organisation

From the outset, it was agreed that each team member would be responsible for development and research into board generation techniques. Additionally specific roles were assigned to each team member to oversee particular aspects of project delivery, these were: project leader, communications lead, development operations (DevOps) lead, project manager and documentation editor.

To ensure that the produced code was compatible with Jumanji's framework, the team forked the code repository and performed all its activities in a self-contained way by building a board generation package at the same level as the Jumanji package. This ensured that no dependency issues were inadvertently created. The team agreed to use Google style docstrings and typing in code, small commits and the use of pull requests. This was in keeping with the style used in Jumanji and helped to maintain high code quality and standards.

Over the term, the team met with the Industrial Supervisor weekly and presented a slide pack summarising activity progress. A set of priorities (features) for the next week was also agreed in this meeting. The term was primarily spent on research and development of board generation techniques including subsequent improvements and fine-tuning. Additionally, benchmarking and board testing environments were implemented. The team also created a visualisation script (built over Jumanji's visualiser) to help save and monitor renderings of visual boards.

Over the Easter period, the team received the RL agents from InstaDeep which would be used for training and testing. As a result, the team's focus shifted to generator testing and solution conversion to JAX. Through this work, the final `SeedExtension` generator was designed to benefit from the observations made from testing the prototype generators.

2.6 Working Practices

The team adopted some industry best practices in order to maintain an effective working style. In particular, Git workflow [9] and Agile [8] principles were used.

Firstly, principles of Git flow and Feature Branch workflow were adopted. Notably the branch structure was divided into `main`, `dev` and `feature` branches, which particularly suited for release cycles. The full benefit of this feature was not realised as working on a fork of the Jumanji repository meant that updating the main branch could cause potential confusion at the end of the project. This was due to the fact that the final deliverable into Jumanji would be a subset of the code developed over the course of the project. Hence the final working style would be better described by the Feature Branch Workflow where the `dev` branch served as the `main` branch throughout the majority of the project. Additionally when each feature branch was ready to be merged, it was first rebased onto the most recent version of `dev` to prevent any merge conflicts. Furthermore any merge was preceded by a pull request (PR) approved at minimum by the team leader via a PR review or via other means.

Secondly, elements of agile were used to support the ways of working. Since the Agile methodology is underpinned by the principle of valuing people over processes, the team selected the elements of the Scrum [18] methodology that aligned best with the working style. As such during the term the

focus was on sprints, with task control and time tracking done less formally via an excel spreadsheet. However, when the team moved to work full time on the project, daily stand ups were used and tasks were divided into epics and tracked via Jira [2]. Initially story points were also used to track individual workloads to ensure an even distribution, however as last minute requirements came into play these were removed since this provided more administrative work than benefit to the team.

3 Research Approach and Methods Summary

3.1 Solution Research

The team made the strategic decision to develop the initial board generators' logic in NumPy. Due to its wide scientific use and familiar syntax, NumPy is well-suited for fast prototyping, easy debugging and logic evaluation. Additionally, JAX has many functions and operations similar or identical to NumPy's making the eventual transition easier. As an initial proof of concept, a generator based on a random walk was developed as a baseline. This was because it was explainable, intuitive and provided a good grounding for understanding the Jumanji environment. Following this, the team carried out more research into maze generation and procedural generation techniques. Out of this, methods based on Breadth First Search (BFS), L-Systems, Wave Function Collapse (WFC) and Number Link were developed. Following learnings during the conversion to JAX, two further board generators were created that combined elements from the NumPy prototypes.

3.2 NumPy-Based Generators

Random Walk

This implementation randomly placed wire heads on the board and then allowed each wire to walk randomly until it hit a maximum length or no more moves were available. This was done sequentially to allow initial wires to work as obstacles. During development, constraints were added to the random walk making it biased so that it would not produce invalid behaviours as discussed later in section 4.1.1. Additionally it was used as a blueprint in implementing robustness features such as ensuring a user could not request more wires than cells on the board. Following from this, exceptions were defined to flag undesired behaviours such as invalid number of heads/targets on board, invalid wire structure or a mismatch in the number of placed wires against those requested.

Breadth-First Search

The Breadth-First Search (BFS) method came from reconsidering the problem as a solving rather than a generation problem. It was noted that a number of maze solving techniques employ some elements of BFS [17]. For each wire requested, a head and target position are selected at random and BFS is used to "solve" for a path from head to target within the available cells on the board. As with the Random Walk method, the initial wires act as barriers that subsequent wires need to avoid.

It was noticed that BFS boards primarily contained long wires with few bends. The solution to this problem was **clipping**. The algorithm, assesses and removes some wires on the board based on a heuristic, then attempts to fit new wires. The heuristics used included: length-based (removes the *shortest* or *longest* wires), bends-based (removes set number of wires with fewest bends or all the wires with fewer bends than a defined value (*min bends*)), other heuristics (removes wires based on a first-in-first-out (*fifo*) basis or at *random*) and combined methods (using multiple aforementioned metrics by applying rounds of filling and clipping for a preset number of loops or until certain *thresholds* were reached). With the various clipping methods incorporated, it was noted that more challenging boards were produced. Particularly for boards with wires greater than the number of rows or columns.

L-Systems

An L-Systems[3] board initialises either with starting points only or length-2 head and target wires.

[3]L-systems were introduced by biologist Aristid Lindenmayer [12] as a way to model the growth of plant cells. They have since been widely used in application to procedural generation for creating complex structures and intricate patterns.

A globally applied set of rules is then applied for a user specified number of iterations to fill the board. The rules (actions) available to each wire, chosen for board diversity, are: push (extend wire), pull (shorten wire), or none (no action). These apply in a probabilistic manner, according to user-specification.

High stochastic action dimensionality of L-systems means wires fill in a way akin to Brownian motion, thus producing wires with many knots and low head-to-target distance. This results in larger boards having wire clumps instead of interleaving. Post-processing techniques (4.1.2) were applied to counteract this. Board generation was possible in NumPy, however, compile time was an issue when using JAX, due to high recursion complexity. Therefore working with non-trivially sized boards was not possible in the JAX re-implementation and as such it was not delivered as a viable solution for Jumanji.

Wave Function Collapse

Wave Function Collapse [11] is a popular method of procedural generation, often found in the random generation of level designs in video games. It is based on creating tilemaps of allowed behaviours within a board to be used as the component pieces, as well as rules for how they can be combined. Tiles are placed on the positions of the board with minimal entropy. Tiles are added successively in this manner, backtracking if necessary, until a board is created. This mirrors wave function collapse in Quantum Mechanics: starting with many possible combinations of states, variables are observed one at a time until left with a single observed combination of states [20]. The final observed state here is the board.

Although initially promising, two major limitation were that sampling weights required tuning for any board size/number of wires combination and that a fixed number of agents could not be defined. The latter was overcome by post-processing but given the lack of flexibility of this method, it was not well suited for this problem.

Numberlink

The last generator attempted was derived from an open-source generator [1] for the game known as Number Link. This was tested as it was already known to produce potentially difficult boards and visually, the board generated could mimic ones similar to PCB boards. It combines some aforementioned ideas such as a biased random walk, but also relies on more complex concepts such as creating a dual puzzle on a board twice as large as the desired grid before shrinking it down.

The open-source generator was adapted to allow a specification of a fixed number of agents to meet Jumanji's requirements. With this feature when compared visually and via the RL agent it provided some of the more difficult boards. However the use of this repository, was limited by its AGPL [7] license. This meant that if used within Jumanji, the whole library would have to follow this license which is more stringent than the existing Apache licence [19]. Therefore this solution was only used as a research benchmark.

3.3 JAX-Based Generators

3.3.1 Sequential and Parallel Random Walk

The first solution converted to JAX was the `RandomWalk`, as it was similar to the `UniformRandomGenerator`, albeit with solvability, but with logic that did not contain recursion. This made it well suited to a rapid JAX re-implementation. The random walk can be implemented in two ways: *sequentially* i.e. whole wires are placed one after the other, or in *parallel* i.e. where one step of each wire is placed at the same time. The former allows initial wires to act as obstacles potentially making the following wires more interesting, but it can also lead to a large variance in wire length. The latter is more likely to produce wires of similar lengths, however instances where two wires opt to move into the same cell need to be appropriately handled in the implementation.

Both approaches were implemented in JAX, as the sequential approach followed logic from the

`RandomWalk` generator in NumPy and the parallel built on the logic of the random policy rollout in Jumanji. Ultimately the parallel solution was selected as it took advantage of the existing data model within the package allowing for easy maintenance. Additionally, operation parallelisation meant that it was also significantly quicker to compile and generate boards as discussed in Section 5.

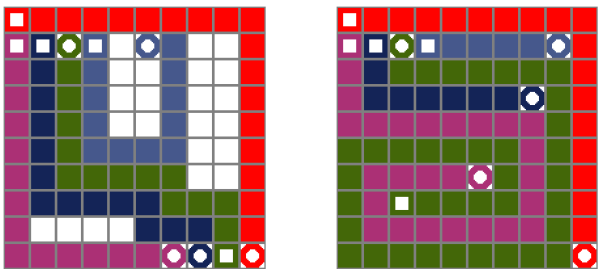
3.3.2 Parallel Random Seed Extension

As highlighted in section 3.2, there were aspects of each generator that contributed to interesting behaviours in the boards. Therefore these findings were used to design the final generator, which combined them into one coherent and flexible framework. Here the generation process relied on placing the head and target of each wire as length-2 head and target pairs (random but adjacent locations on the board) and then extending these wires in parallel (as defined above) according to the rules set out by the hyperparameters summarised in Table 1.

Hyperparameter	**Role and Description**
Two-sided extension	Move both target and head in extension process (two-sided) or just target (one-sided).
Randomness	Determines behaviour of wires during extension stage. With value 0, the current direction is followed until no empty cells are available then a new direction is randomly selected. The randomness value (0 to 1) is the probability that any individual step will be overridden by a random direction.
Extension steps per wire	Maximum steps to be taken by each head/target cell in each extension stage. With value 0, no steps are taken. By default, this value is 10^{23}, so the wires will effectively continue extending until no more moves are available.
Number of cycle iterations	Number of times BFS optimisation and extension cycle is applied.

Table 1: Hyperparameters of `SeedExtension` generator and their role in determining generation behaviour.

The two key process steps at play are the extension stage and the optimisation stage, which can be cycled through any number of times. The former extends existing wires into empty locations and the latter performs BFS optimisation to remove invalid wire elements and thus free space for the next extension stage. Figure 2 visualises the change in wire behaviour over 8 iterations of optimisation applied to a board.

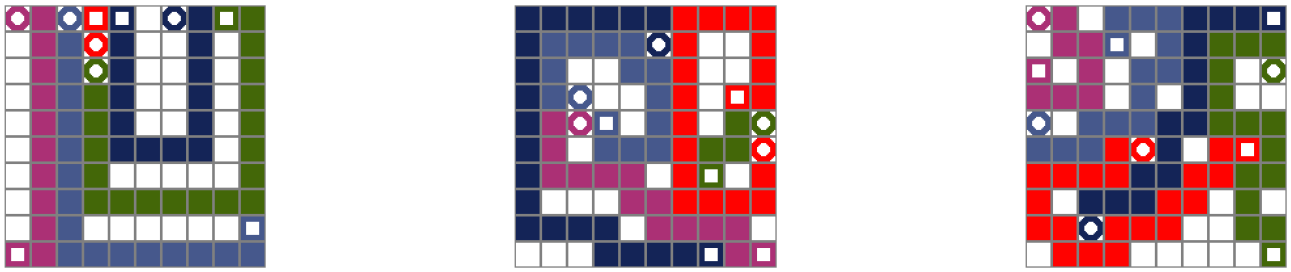


(a) 1 Iteration **(b)** 8 Iterations

Figure 2: Iterative application of wire-extension and minimum-length optimisation

The change in board behaviour based on the *randomness* parameter, are visualised in Figure 3, where extension was applied once (i.e. no BFS optimisation is added thus resulting in non-direct solutions within the boards). When combined with the optimisation step, this then results in varying board behaviours.

The key findings from experimentation with the hyperparameters are as follows. A two-sided extension results in faster generation time. Board complexity can be increased by increasing the number of steps



(a) Randomness=0 **(b)** Randomness=0.5 **(c)** Randomness=1.0

Figure 3: Seed extension boards using one extension with varying randomness values.

taken or the number of extension-optimisation cycles, but complexity plateaus after about 8 cycles. When using a large number of iterations (21) and testing against the trained agents in Figure 13, it was found the the randomness only produced a slight difference in the number of wires connected (2.5%), suggesting that randomness only plays a large part when few iterations are used.

3.4 Solution Comparison

As outlined previously, the problem statement required the boards generated to be solvable and difficult. The former was ensured through the inherent design of the board generators, however the latter required the development of comparison metrics. This was done in three ways: static evaluation of solved boards, testing performance of an agent rolling out a random policy and using a reinforcement learning (RL) agent. The RL agents used were provided by a Jumanji release in March, and as such initially only the first two methods were used.

3.4.1 Random and Reinforcement Learning Agents

Implementing a random agent acted as a first measure of difficulty, as a good performance would suggest the boards were too easy for an RL agent. This was first implemented in order to flexibly fit in with the board generation prototyping, and to suit the data collection requirements. Following the Jumanji release in March, the benchmarking implementation was refactored to use the environment's random agent to ensure compatibility. The data collected was total reward, proportion of wires connected, length of wires and number of steps taken to termination. The agent was generally unable to make more than 1 connection (if any) with the results showing that connections were less likely when the head and target were further away. The results will not be reported as they were consistently poor on all the boards and did not provide further insights to differentiate or quantify the boards.

Training an RL agent using the board generators then allowed the comparison of learning curves to understand how many wires the agent was connecting over time. The working hypothesis was that more difficult boards would cause the agent to learn more slowly and likely accomplish a fewer connections at the end of training. The approach to this method is discussed further in section 4.3.1.

3.4.2 "Static" Board Comparison

The generator design allowed solved boards to be obtained, which was used to collect metrics to quantify board difficulty without the need for an RL agent, which is referred to as a *static* comparison. Some metrics could be directly collected from a solved board, such as wire length, number of bends per wire, distance between the wire heads and targets and proportion of the board filled. However these on their own could not illustrate complexity, therefore two novel metrics were designed: number of detours taken by wires and the variety of behaviours around the grid.

Number of Detours

This was defined by how often a wire appears on two opposing sides of a head or target (either left/right or top/bottom). The expectation was that solved boards with more detours would be harder to solve.

However, some detours proved to be extraneous which did not increase board complexity, and the number of detours did not provide a higher correlation with the number of wires connected than the board density did. As a result, although this measure was collected, it was only used alongside other metrics. An example of this is shown in Figure 12 in the Appendix.

Behaviour Diversity

To investigate the variety of behaviours in a board, inspiration was taken from applying filters to images to summarise the behaviour of the neighbours of a cell. As such, first each board was scored and then cross-correlation using Gaussian Kernel was applied to each cell. To add more meaning to the scoring, a multiplier was used which was computed from the number of wires within the kernel window, as a higher number of wires interacting was likely corresponding to a more difficult area in the board. A diagram of this process in shown in Figure 4. The scores assigned to each cell were based on an assessment of perceived difficulty i.e. an empty cell would not add interesting behaviour and therefore had a negative value, whereas the filled cells were of interest. Then a higher score was given to the heads/target as they are more scarce.

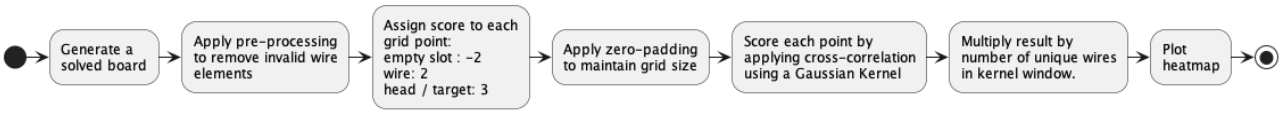
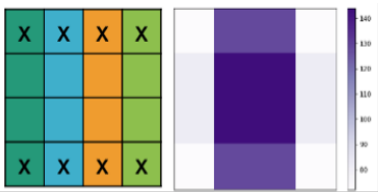


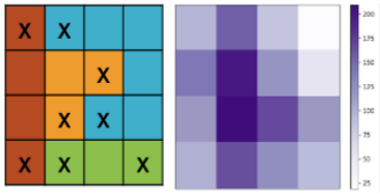
Figure 4: Procedure for generating heatmap representation of a board.

This allowed a *heatmap* to be generated for any board, an example for a single board is given in Figure 5. Despite providing a good visual representation, a single metric was desired to be able to compare the generators numerically. There the principles of Quality Diversity Algorithms [10] were used for inspiration. The quality element was provided by the individual scores, while diversity was from the number of different scores within a board was computed. This aimed to estimate how many different behaviours an RL agent might experience within training, with the expectation being that more diverse boards would be more difficult for the RL agent to solve.

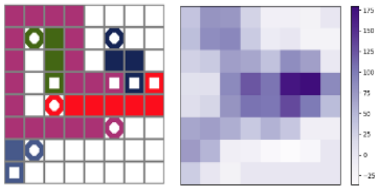
To illustrate the essence of this concept to inform board difficulty, Figure 5(a) and (b) presents heatmaps for a simple and more complex board respectively. The simple board, has few differences in behaviour as compared to the more "complex" board. This method was then used to average scores over n boards to compare high-level behaviours between the boards as discussed in section 5.



(a) Simple board with heatmap showing 3 unique behaviours



(b) More complex board with heatmap showing 8 behaviors.



(c) Board generated using `BFS` generator and associated heatmap.

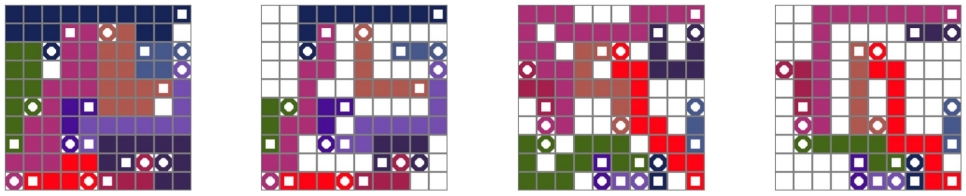
Figure 5: Example of heatmap generated for a two sample boards and one realistic board generated.

4 Technical decisions, design and architecture

4.1 System Design

4.1.1 Common Board Processing

Using learnings from both the `BFS` and `RandomWalk` board generators, a common `BoardProcessor` class was created to ensure that static board statistics were calculated in a consistent[4] way. Given any board object or a NumPy array, the class is able to do the following: verify that the solved board is valid, refit any wires that were not acyclic and calculate statistics about the wires on the board for collation. These included wire lengths, bends and proportion of the board filled. The class also performed processing tasks like shuffling heads and tails, shuffling the wires' encoding or removing any wires if required. It was noted that some methods from this class would be useful during board generation. Therefore they were incorporated into some of the board generators and developed into board enhancement techniques detailed in section 4.1.2.



(a) `WFC` Board with invalid wires

(b) Processed only valid wires

(c) `RandomWalk` board with superfluous paths

(d) Solved board with minimal paths

Figure 6: Examples of the `BoardProcessor` board improvements

4.1.2 Board Enhancements

Two techniques were developed as they were shown to increase the difficulty of boards after they were generated: wire thinning and wire extension. *Wire thinning*, relied on concepts of valid wire connections (true if and only if there is a single path from the head to the target within the wire cells) and BFS paths from the head to target being minimal distance paths [21]. For boards that did not generate valid wires, a BFS-based technique was applied to thin the wires during generation. This removes superfluous wire paths and allows for more available space for wires generated later on. *Wire extension* was based on the hypothesis that boards would be harder to solve if wires were forced to extend around other wires. Inspired by `RandomWalk` and `LSystems` techniques, wire extension was applied in post-processing and extended the head and/or target of each wire into available valid cells.

4.2 Benchmarking

The benchmarking package was designed for flexibility early on in the project to be able to handle new metrics, generation for any generator etc. Thus a clear separation of concerns was applied within the code with some classes providing data interpretation e.g. via visualisation, and others providing the data model and facilitating data handling.

This structure was then used to provide the user with two simple scripts to perform the desired process, with features including performing the simulation (or benchmarking from saved data files), saving the data generated, saving the plots, specifying the desired board generators or running a benchmark on all generators listed in the `BoardName` class discussed in section 2.3.

[4] As board solutions are not unique, the desire was for statistics to be calculated on solutions as similar as possible. For consistency when collating statistics for different generators, wire paths retraced to make them minimal and acyclic

4.3 RL Training Pipeline

The RL agents provided were based on a Convolutional Neural Network followed by a transformer to allow optimising the multiple agents (representing multiple wires) simultaneously. Due to the computational expense, training on a GPU was required. As such the Imperial SLURM cluster was used to allow easy job submission and management, as well as to benefit from the pool of NVIDIA GPUs. To allow the team to work at speed across multiple workstreams, the following were essential: ability to quickly run experiments and to version the code easily to minimise potential errors when submitting jobs.

The code was versioned using the package poetry [16] to build python wheels that allowed the code to be installed in one step on the virtual environment within the cluster. This avoided the need to clone any repositories or concerns about what features were available in the current codebase.

To facilitate job submission, a folder `agent_training` was built, which contained all files required to run a job including the python training script, a training README (with a cluster and experiment set up guide), the bash training script and required configuration files. This could then be copied securely onto the cluster and used from there. This implementation allowed for both team members and future researchers to easily set up training.

To log the results of the experiments, the Neptune logging tool [13] was used as it enabled online real time tracking of experiments and was a feature provided by Jumanji. As such the Neptune API token needed to be passed as an environment variable to allow the connection. Since this is sensitive to each user, the best way to use it within the cluster was considered. Firstly, a docker image was built as it would allow passing the environment variable into the container at run time and would not be stored within the cluster at all. This solution also removed any versioning concerns. However, Imperial's GPU cluster does not allow the use of docker due to size and permission considerations. Therefore, API keys were defined within the bash script for running the job, which meant the token would be deleted from memory as soon as the job was complete.

4.3.1 Working with the RL agents

Once the RL agents were provided by InstaDeep, initially combining them with the NumPy generation was attempted, but due to JAX caching the generators were not capable of producing board online during training. Generator selection for JAX conversion relied on insight from RL training, therefore alternative methods of obtaining these were designed. Firstly, online training using the existing infrastructure was followed by offline evaluation of the agent using boards generated using the NumPy implementations. Secondly, an offline board generator was implemented. It created a dataset of boards in the compilation stage of training which could be sampled from randomly in training and evaluation. These approaches provided an idea of performance and allowed prioritising development, as outlined in section 5. This was performed on a fixed board size (10x10) and number of agents, 5. An investigation into varying these parameters was considered but not undertaken due to the existing large scope of the project.

4.4 Conversion to JAX

A stretch goal of the project was to contribute a generator in JAX into Jumanji. As previously discussed in sub-section 3.1, generators were developed in NumPy. The subsequent porting over of logic from NumPy to JAX was not without major challenges, as summarised in Table 2 below.

Challenge	Description
The Immutability of JAX Arrays	JAX arrays cannot be modified in-place which many generators relied on, notably the clipping methods inherent to the `BFS` generator. As a result this generator was not converted to JAX.
Inability to use Dynamic Indexing and Arrays	JAX is designed to work with arrays of fixed shape known at compile time and does not allow indexing with values that are not known at that stage. Hence dynamic arrays lead to errors during run time and had to be vectorised as much as possible.
Inability to use recursion and for-loops or while-loops	JAX does not support recursion well, nor conventional Python for loops or while loops which are central to many of the generation methods, albeit providing JAX-specific functions to perform as work-arounds. As a result the `WFC` generator, which relied heavily on recursion, was not converted
Difficulty in Debugging	JAX code can be more difficult to debug due to its use of lazy evaluation and deferred execution making it harder to pinpoint the source of errors.

Table 2: Challenges in JAX conversion of board generation prototypes.

To tackle these challenges, the board generators were evaluated to ascertain the viability of their JAX conversion. The `RandomWalk` generator was converted to JAX as a Minimum Viable Product (MVP) as it is relatively simple to enact but produces complex and solvable boards. It also does not suffer from the issues highlighted for other generators in Table 2. The L-Systems generator was also converted to JAX but its memory leak issues[5] meant that it would not be a viable generator for Jumanji.

To minimise compile and running times for the random walk, as mentioned in section 5, wires were generated in parallel rather than sequentially. This also produced more difficult boards as compared using static metrics. Thus resulting in the `ParallelRandomWalk` as the final deliverable. To satisfy an additional client request for a more complex generator the `SeedExtension` generator was implemented in JAX, but the delivery of a production level ready version was not in-scope of the project. Therefore a version that could be imminently ready for production was delivered as part of the research repository.

5 Evaluation

5.1 Generator Comparison for Solution Selection

The evaluation is bipartite: a discussion on which generators produced the most difficult boards and one on how well the final implementation met InstaDeep's requirements. A summary of features implemented by all generators is shown in Figure 7. Notably, solvable and difficult generators were delivered, some of which also enabled training an RL agent to learn complex policies, thus meeting all project requirements.

[5]Memory leaks occur when programs allocate increasing memory over time without releasing it back to the system. In JAX, they can occur when intermediate results are required to be stored and not released until the end of the computation. This can lead to out-of-memory errors or slower performance over time as more memory is consumed. [6]

	NumPy					JAX	
	Random Walk	BFS	L-Systems	Number Link	WFC	Parallel Random Walk	Seed Extension
Generate Solved boards	✓	✓	✓	✓	✓	✓	✓
Generate training boards	✓	✓	✓	✓	✓	✓	✓
Solvable	✓	✓	✓	✓	✓	✓	✓
"Difficult"	✓	✓	×	✓	×	✓	✓
Offline dataset generation	✓	✓	✓	✓	✓	✓	✓
RL training on Offline dataset	✓	✓	✓	✓	✓	~	~
Online training	×	×	×	×	×	✓	~
Static board evaluations	✓	✓	✓	✓	✓	✓	✓

Figure 7: A summary of capabilities of each of the generators. Green represents full capability, red that a feature is not available and orange that a feature is partially available. In offline training using JAX generators, albeit functional, overfitting was observed due to dataset size. Online training using `SeedExtension`, training was possible, but impractically slow.

5.1.1 Static Boards Comparison

The static board comparison analysis was done in two stages. Heatmaps were compared to eliminate methods with any bias and then static metrics were compared to identify other trends.

From the heatmap comparison in figure 8, it was noted that `WFC` clearly had a bias in placing pins, that random walk more uniformly covered the whole board but the behaviour were not necessarily as interesting as one observed in the variations of the BFS generator. Behaviour of the clipping algorithms is well illustrated through this experiment. Intuitively `BFS` with shortest or minimum bends wires removed have the highest scores as effectively wires with fewest interactions are removed. The reverse is illustrated by `BFS` with longest wire removed and FIFO heuristics.

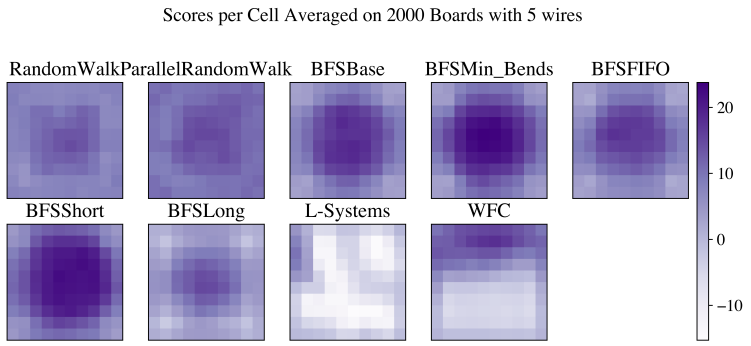


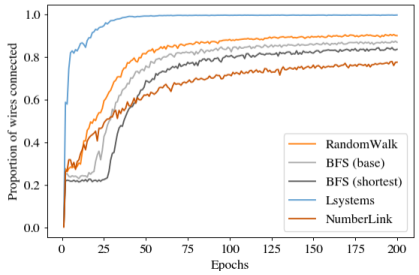
Figure 8: A summary plot showing heatmaps averaged for 5000 boards per generator. `NumberLink` is omitted here as its solved boards aimed to fill the whole board, thus artificially increasing the scores even if a simpler solution was possible.

Based on the evaluation of static metrics `RandomWalk`, `BFS`(*short*) and `NumberLink` were shortlisted as potential solutions. They were found to have the largest number of bends per wire, largest head-target Manhattan distance and most visually interesting[6] boards respectively. These results are reported in Appendix A. `LSystems` combined with post-processing was also pursued in the initial JAX conversion, but as explained previously it was found to not be a viable solution.

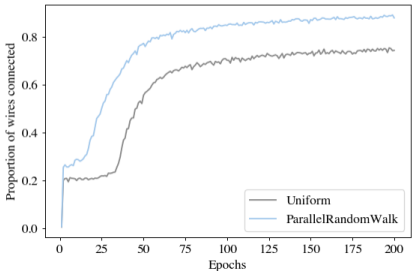
[6]Visually interesting boards have wires with a diversity of behaviours as demonstrated by the heatmaps with multiple wire interactions

5.1.2 Evaluation using RL agents

The first two steps to use the prototype generators were to train agents using an offline dataset and to evaluate an agent trained on the `UniformRandomGenerator` on boards generated using the implementations. Note from the learning curves in Figure 10(a) that all generators achieve a better performance than the baseline generator suggesting that they are easier to solve than the base solution. This however can be attributed to two reasons: the fact that the base solution does not produce solvable boards thus inhibiting the agent from learning better policies and that there might be some degree of overfitting to the offline dataset. The former hypothesis is supported by the fact that the `ParallelRandomWalk` achieves a higher proportion of wires connected than the `UniformRandomGenerator`, as shown in Figure 10(b), but the `ParallelRandomWalk` should be at least equivalent and is likely to be more difficult. The best performance was achieved by `Numberlink` but, given the licensing considerations outlined previously, it was not a viable end solution.



(a) Evaluation results of agents trained on offline datasets created using NumPy-based generators



(b) Evaluation results of agents trained online on JAX-based generators

Figure 9: Ratio of connections for an agent trained on a Uniform generator

	`BFS` Base Board	`BFS` Short Board	`BFS` Long Board	`Sequential RandomWalk` Board	`LSystems` Board	`NumberLink` Board	`Seed Extension` Board
`Uniform` Agent	**4.56**	**4.43**	**4.72**	**4.67**	**4.98**	**4.04**	**3.86**
`BFS` Base Agent	4.42	4.27	4.63	4.54	4.97	3.75	3.54
`NumberLink` Agent	4.34	4.22	4.57	4.48	4.93	3.64	3.49
`LSystems` Agent	3.23	2.95	3.63	3.51	4.88	2.74	2.50
`Sequential RandomWalk` Agent	4.32	4.16	4.56	4.51	4.96	3.64	3.47
`Parallel RandomWalk` Agent	4.38	4.22	4.57	4.52	4.96	3.74	3.55

Table 3: Average connections for agents trained on boards from one generator (in rows) and evaluated on boards generated by each of the generators (in columns).

In Table 3, agents trained on different generators are evaluated on boards created by the other approaches. This is to understand how the policies learned might generalise to other boards. (`Uniform` boards were not evaluated because they are not guaranteed to be solvable.) It shows clearly that the agent trained on the `Uniform` boards (bold numbering) is the strongest no matter which board is used for evaluation, while the `LSystems` agent is the weakest. It also shows that the boards generated by the `RandomSeed` generator (in red) are the most difficult to solve for any of the agents, followed closely by the `NumberLink` boards (in yellow), despite the fact that some `Uniform` boards have no solution

which connects all the wires. Thus these boards are a definite improvement over the original boards in solvability while also being difficult.

`LSystems` boards proved to be the easiest (almost always being completely solved). There was a definite correlation between the difficulty of boards an agent was trained on and that agent's aptitude on all the boards. For example, in earlier experiments comparing parallel vs sequential random walk strategies, it was found that the `Parallel Random Walk` boards were slightly more difficult than the latter, and here the `Parallel Random Walk` agent is slightly more successful on the various boards than the `Sequential Random Walk` agent.

5.2 Final Implementation Evaluation

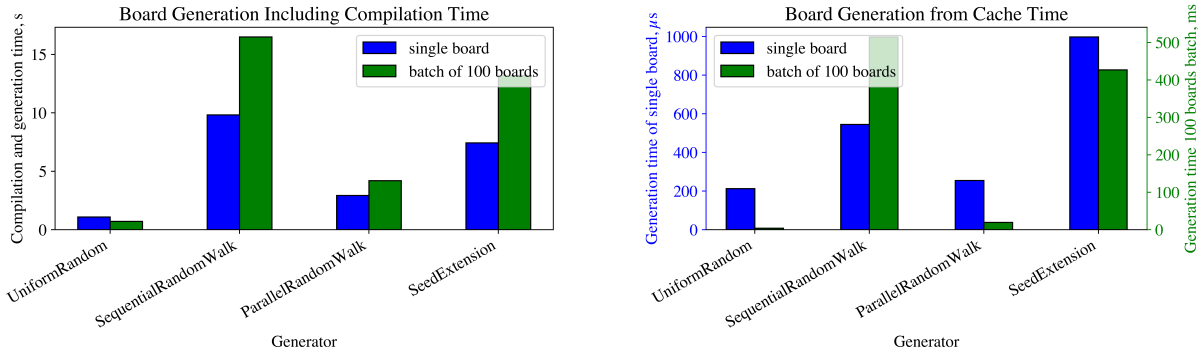
It is important to note that the final implementation was originally scoped to be done in NumPy, but scope of the project was expanded into delivering the product in JAX due to efficient prototype delivery. Therefore the final deliverable became a JAX-based generator and a research repository containing NumPy prototypes and benchmarking capabilities. As mentioned in section 4.4 the conversion to JAX posed many challenges that limited the amount of features that could be implemented in the project time frame, as such an MVP solution was delivered to ensure a solvable generator could be merged into Jumanji to enhance the library. Then a second JAX-based generator was built as the second stretch goal for the project.It met compilation time but not the runtime requirements. Since it was not within the scope of the project to deliver this as a contribution to Jumanji, it forms part of the research repository.

5.2.1 Compilation and Running Time Speed Comparisons

One of the challenges encountered was compilation time of the JAX Generators. This was also correlated with the run time per epoch when training an RL agent. During training, compilation is performed once at the first epoch. As such it is desirable for this to be as quick as possible, but not necessary for later training. Time to generate a board in cache is more important as it will be a cost repeated in training.

When comparing the `Uniform` and `RandomWalk` generators, the `ParallelRandomWalk` both compiles and runs in half as much time as the `SequentialRandomWalk`. This alongside the difficulty analysis from 5.1.1 made the `ParallelRandomWalk` best suited for a contribution into Jumanji.

When considering the `SeedExtension` generator, the version with no BFS optimisation is roughly twice as slow as the `ParallelRandomWalk`. Using cache it is 7.8 times as slow as the baseline as opposed to 3.5 times for the `ParallelRandomWalk`. This corresponded to a non-viable extension of training time.



(a) Board generation with compilation time.

(b) Board generation from cache

Figure 10: Board generation times for single and batch of 100 boards including compilation time (a) and from cache (b). Run on 3.3 GHz Dual-Core Intel Core i5.

The `SeedExtension` generator was also considered in a standalone fashion to compare the impact of varying the randomness parameter and the number of extension-optimisation cycles. The randomness parameter provided negligible change in compilation with 1 iteration however it caused a 20% increase in compilation time for 10 iterations when increased from 0.5 to 1. Interestingly when generating boards from cache this trend was not followed. Here boards with a randomness of 0.5 resulted in the highest generation times. Overall although there are differences in generation these are negligible as compared to much quicker speeds achieved by the `ParallelRandomWalk` generator. This analysis would need to be revisited when the `SeedExtension` solution was optimised to allow a viable speed comparison to be performed.

	Compilation and board generation, ms		Board generation from cache, µs	
Randomness	1 iteration	10 iterations	1 iteration	10 iterations
0	5660	5390	704	700
0.5	5580	5490	772	987
1	5360	6610	703	883

Table 4: Compilation time for board generation of a single board using the `SeedExtension` generator for randomness 0, 0.5 and 1 compared for 1 and 10 iterations of extension-optimisation cycles.

5.2.2 Parallel Random Walk - the MVP Deliverable

To ensure robustness of the solution this generator was fully unit and integration tested. The test coverage for this module was verified using the `coverage` [3] package, with 100% achieved. The pull request has passed all checks and is pending approval from the InstaDeep team. Overall, although this solution is not the most advanced generator developed, it provided a direct improvement on the existing implementation both on solvability and difficulty, with performance satisfactory to allow an imminent merge into production.

5.2.3 Seed Extension - the Extended Deliverable

This final implementation was delivered as part of the research repository, but could be rendered production-ready quickly if more time was available for delivery. As show in sample figures 3 and 2 this generator is capable of producing boards following a variety of rules, thus enabling a tuning of difficulty. Although online generation was possible, the implementation was not parallelisable enough thus causing unacceptable training times. Tuning and comparison of difficulty of as a function of the hyperparameters is outside of the scope of this report, but could be added to to the existing testing suite within the research repository.

5.3 Meeting User Requirements and Open Source Contribution

The key requirement of the user (InstaDeep) was to receive one validated, good and solvable board generator for the routing problem, based in Python. The additional requirement of upon successful completion of the key requirement was to implement a generator in JAX and contribute it to the Jumanji library. Over the course of the project, the team researched and produced and validated five board generators [7] in NumPy that produced complex boards. Solvability was ensured at generation because the boards were generated as solved boards in all techniques. In addition to this, the team further developed two board generators based in JAX[8]. In total producing seven validated, good and solvable board generators. This was cap-stoned with a contribution of the team's JAX generators to the Jumanji Environment.

[7] `RandomWalk`, `BFS`, `WFC`, `LSystems` and `NumberLink`
[8] `ParallelRandomWalk` and `SeedExtension`

5.4 Limitations and Possible Expansions

The major challenge that would be the focus of potential future work is the improvement of the paralleisation of the `SeedExtension` solution to make it viable for generating boards online with RL agents. This final optimisation was attempted but given the time and scope of the project this was not completed.

Furthermore as seen in section 5.1, the clipping methods i.e. removal and re-insertion of certain wires could also impact the difficulty of boards, therefore this could also be a desired feature in future releases. With these in mind, further work could also be undertaken on training agents using varying parameters and measuring performance in more complex scenarios to observe whether useful policies were learned. Additionally combination with an environment looking to optimise 2D component placement could be an interesting future project.

6 Conclusion

In conclusion, the objective of delivering a board generator producing solvable and difficult boards for the routing problem was accomplished through the delivery of 7 generators, 1 of which was fully production ready and awaits merge approval from the InstaDeep team and 1 with some support regarding code parallelisation could also be productionised easily. The remaining generators were delivered as part of a research repository. The investigation into what make a difficult board was attempted through training of reinforcement learning agents as well as through computation of static metrics. The challenge remains that although local metrics provide some insights, understanding the global difficulty of the board remains non-trivial.

Capabilities for benchmarking were provided as well as a guide to running experiments on SLURM clusters for potential research users of the repository.

References

[1] Thomas Dybdahl Ahle. Numberlink: Program for generating and solving numberlink / flow free puzzles, 2021.

[2] Atlassian. Jira Software (Project Management Tool), accessed 2023.

[3] Ned Batchelder. Coverage.py, 2005–2021. Accessed: May 2, 2023.

[4] Clément Bonnet, Daniel Luo, Donal Byrne, Sasha Abramowitz, Vincent Coyette, Paul Duckworth, Daniel Furelos-Blanco, Nathan Grinsztajn, Tristan Kalloniatis, Victor Le, Omayma Mahjoub, Laurence Midgley, Shikha Surana, Cemlyn Waters, and Alexandre Laterre. Jumanji: a Suite of Diverse and Challenging Reinforcement Learning Environments in JAX, 2023.

[5] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[6] Douglas Crockford. JavaScript Memory Management, 2012. Accessed: April 30, 2023.

[7] Free Software Foundation. GNU Affero General Public License, 2007. [Online; accessed 29-April-2023].

[8] Martin Fowler. The New Methodology. 2000.

[9] GitLab. GitLab Flow, Accessed 2023.

[10] Daniele Gravina, Ahmed Khalifa, Antonios Liapis, Julian Togelius, and Georgios N. Yannakakis. Procedural Content Generation through Quality Diversity. *CoRR*, abs/1907.04053, 2019.

[11] Maxim Gumin. Wave Function Collapse Algorithm, 2016.

[12] Aristid Lindenmayer. Mathematical models for cellular interactions in development II. Simple and branching filaments with two-sided inputs. *Journal of Theoretical Biology*, 18(3):300–315, 1968.

[13] Neptune.ai. Neptune: Cloud-Based Machine Learning Experiment Tracking.

[14] Ed Pegg Jr. Beyond Sudoku. *Mathematica Journal*, 10(3):469–473, 2007.

[15] Real Python. Factory Method in Python, accessed on 2023-04-26.

[16] Python Packaging Authority. Poetry: Python dependency management and packaging made easy.

[17] Adil M.J. Sadik, Maruf A. Dhali, Hasib M.A.B. Farid, Tafhim U. Rashid, and A. Syeed. A Comprehensive and Comparative Study of Maze-Solving Techniques by Implementing Graph Theory. In *2010 International Conference on Artificial Intelligence and Computational Intelligence*, volume 1, pages 52–56, 2010.

[18] Scrum.org. What is Scrum? (an agile framework), accessed 2023.

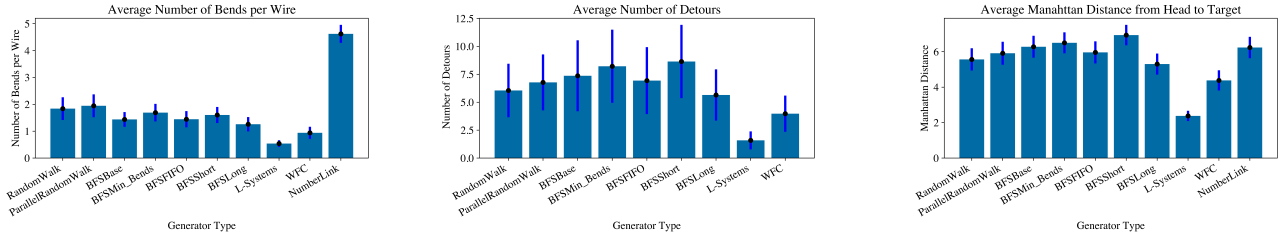
[19] The Apache Software Foundation. Apache License 2.0, 2004.

[20] Wikipedia. Wave Function Collapse. [Online; accessed 25-April-2023].

[21] Jeremy Zhang. Search Algorithm Introduction: Breadth-First Search, January 2019. Accessed: 2023-04-29.

Appendices

A Static Board Statistics



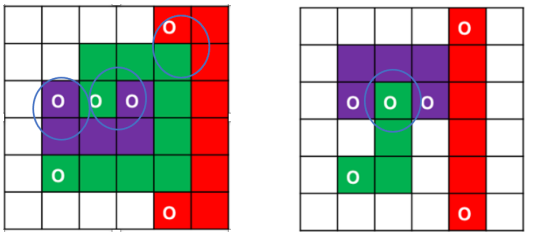
(a) Average number of bends per wire.

(b) Average number of detours per wire.

(c) Average Manhattan distance between heads and targets.

Figure 11: Metrics for boards compared for each generator averaged on 5000 boards. `NumberLink` is not shown on figure a) and b) as its solution method produced starkly different results as outlined in the caption of figure 8. We note that the standard deviation in figure b) is very large, but that for qualitative comparative purposes this was sufficient.

B Number of Detours



(a) Count of 3 detours

(b) Count of 1 detour

Figure 12: Comparison of two results for the count of number of detours - detours annotated by a blue circle

C Evaluation of RL agents

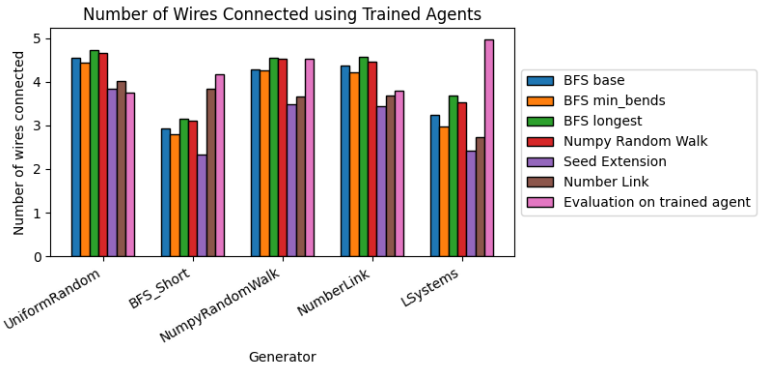


Figure 13: Average connections for agents trained on boards from one generator (bars) and evaluated on boards generated for another generator (rows)