Imposing Constraints on Randomly Generated Graphs in *CatSAT*

Mercedes Sandu

A thesis presented for the degree of
Master of Science in Computer Science

Department of Computer Science
Northwestern University
Spring 2024

Contents

Imposing Constraints on Randomly Generated Graphs in *CatSAT*

Mercedes Sandu

Abstract

Graphs are extremely versatile data structures used to represent pieces of data and the connections between them in a variety of computer science fields. Specifically, in the video game development space, graphs can be used to represent a variety of concepts, from character relationships to waypoint-based maps to branching questlines. Oftentimes, game developers seek to randomly generate such game content to promote replayability, integrate content with mechanics, and lessen the manual design workload in the development process. SAT solvers are logic programming systems that can be used to allow video game developers to specify constraints for the content which they wish to randomly generate. Previous game development research has utilized SAT solvers to generate objects such as characters and settings. However, generating relationship graphs between objects has remained an open problem. In particular, graph connectivity and transitive closure are generally difficult problems for SAT solvers as they are not expressible in first-order logic. Moreover, previous algorithms for connectivity and pathfinding tend to be slower and more expensive than current SAT algorithms. This paper discusses an extension of the embedded C#-based SAT solver, *CatSAT*, for generating random graphs under certain imposed constraints. Furthermore, we present and discuss an approach to using *CatSAT* to generate graphs satisfying constraints including, but not limited to, connectivity, paths between vertices, density, and vertex degree.

1 Introduction

1.1 Background: Procedural Content Generation

Procedural content generation (PCG) is a method frequently used in video games and other software projects to create randomized data algorithmically over a specific domain. Many notable examples of PCG include (but are not limited to) sidescrolling levels, character design, terrain, story, dialogue, and in-game items. While the reliance of a video game on PCG is a spectrum that can be assessed a variety of ways, there are three main factors worth noting [1]:

1. Replayability and adaptability.
2. Relationship to game mechanics and dynamics [2].
3. Player's control over content.

Possibly the most common outcome of using PCG in a video game is increased replayability, as content that varies from playthrough to playthrough of a game can lead to diverse play experiences. Another

benefit of using PCG in video game development is the automation of design processes and bypassing technical limitations, which is especially advantageous for developers working on small teams with limited resources.

PCG can also be intricately and integrally tied to the mechanics and dynamics of a video game. In these cases, the video game directly relies on the PCG system such that gameplay would otherwise not be possible. Notable examples include games that rely on randomized items for the player to choose from in combat and infinite sidescrollers that generate the map as the player traverses it. The core mechanics in such games would not be possible to implement without PCG.

Furthermore, PCG has been used to create video games that dynamically adapt to a player's skill level, changing difficulty and other facets of player experiences in the game either at runtime or offline. The player is hence allowed to "interact" with the generator (albeit, in this case, indirectly) for what may be a more desirable playstyle. This kind of indirect control over content being generated in a game is more common than direct control. Direct control over the generator would likely involve the player being able to interact with some kind of tool that tweaks parameters or constraints passed to the generator.

1.2 Background: SAT Solving

As PCG systems rely on random choices in a finite domain to be made under constraints specified by a designer or developer, constraint programming proves to be a feasible and attractive approach for developing such systems. This is because constraint programming is inherently modular and data-driven, allowing designers to add and remove constraints to a particular generator without needing to modify or develop the underlying algorithm that searches for a model satisfying all specified constraints.

Consequently, Boolean Satisfiability (SAT) and Answer-Set Programming (ASP) are two such algorithms used to solve PCG satisfaction problems [3]. Both SAT techniques and ASP have been extensively researched and optimized, and each have their own sets of benefits and drawbacks. ASP's advantages include a succinct first-order language for describing constraint-based problems, support for pseudo-boolean constraints (for example, "pick n from a menu"), transforming programs into SAT problems, and the use of non-monotonic logic to support minimization. The last item on this list is something of a double-edged sword, as its strength comes at the expense of simple, intuitive syntax and an easy divide-and-conquer debugging experience.

ASP is a more favorable choice for PCG systems that rely on concepts such as reachability, transitive closure, and provability. Notable examples include game generation [4] and level generation [5] that force particular solutions. However, these features along with minimization are not necessarily required for many other types of PCG systems, and using a SAT solver would be more appropriate.

A stochastic local search SAT (SLS SAT) algorithm typically initializes the problem with a random truth assignment and repeatedly "flips" the truth assignment of a chosen variable, meaning to say variables that are `true` are flipped to `false`, and vice versa, until a model (solution that satisfies all constraints) is found [6]. The choice

of which variable is to be flipped changes based on the particular SAT algorithm that is being used.

`GSAT` is a greedy algorithm that chooses to flip a variable that leads to the largest decrease in the total number of unsatisfied clauses. Local minima become a problem as `GSAT` may get stuck flipping variables that do not lead to a better solution state; this is unfavorable when considering that Boolean satisfiability requires finding global optima [6].

Randomness is introduced to decrease the likelihood of the solver encountering local minima in the `WalkSAT` algorithm. This is done via a noise parameter, which is a value from `0` to `1` that dictates the degree of greediness employed when selecting a variable to flip. The greediness ranges from none, indicating selecting a variable at random (according to a uniform probability distribution), to entire, indicating selecting a variable that leads to a maximized decrease in the amount of currently unsatisfied clauses. The value the noise parameter takes can significantly affect performance and behavior of the algorithm throughout the solving process; however finding an optimal noise setting is typically difficult. Heuristics such as `Novelty+` and `R-Novelty+` have been developed to optimize the noise parameter while the solver is running [7].

Other variants of SAT solvers have been developed and studied, which will be analyzed and discussed further in Section 1.4.

1.3 Problem Description

CatSAT is a stochastic solver for an ASP-like language written by Dr. Ian Horswill as an embedded language in C#. After converting a SAT problem into conjunctive normal form, *CatSAT* uses a variant of the `WalkSAT` algorithm to generate a model that satisfies the user-inputted constraints. Prior to starting this project, *CatSAT* could only solve for models when inputted either boolean or pseudo-boolean (float or integer) constraints. The primary aim of this project is to expand *CatSAT* to allow for constraints to be imposed on randomly generated graphs. This is accomplished by creating an API with a small variety of different graph constraints as well as the option for the user to implement constraints of their own.

1.4 Challenges and Previous Work

Randomized graph generation under certain imposed constraints has been previously explored, especially attempting to solve the problem using SAT solvers. For instance, Picat (a Prolog-like language) has been used to model graph synthesis problems such as the *Roadrunner*, *Masyu*, *Shingoki*, and *Tapa* problems, and solve them using SAT. Execution times for these problems of different sizes span a range of anywhere from half a second to almost nine seconds including both translation and solving times [8]. A polynomial time algorithm with $O\left(n^{13/2}\right)$ has also been developed to generate labeled planar graphs randomly on a uniform distribution [9].

Graph connectivity in particular has proven to be a difficult problem for SAT solvers, because connectivity of a graph and transitive closure are not expressible in first-order logic (which SAT solvers require). Transitive closure is a data structure used to represent every shortest path between the vertices of the graph. Specifically, a connected graph is a graph whose

transitive closure consists of every vertex pair, signifying that vertex j is reachable from vertex i for each pair of vertices (i, j) in the graph [10]. In particular, first-order logic has a property of compactness, meaning that a set of first-order sentences has a model if and only if every finite subset of it has a model [11]. The class of connected graphs, however, is not compact [12].

However, note that connected graphs with n vertices can be axiomatized using the Floyd-Warshall algorithm [13]:

$$\texttt{connected}(x, y) \leftarrow c(x, y, V)$$
$$c(x, y, 0) \leftarrow \texttt{edge}(x, y)$$
$$c(x, y, k) \leftarrow c(x, y, k-1)$$
$$c(x, y, k) \leftarrow c(x, k, k-1) \wedge c(k, y, k-1)$$

This, however, generates a problem of size $O(n^3)$ for the SAT solver, which is not particularly favorable. Solution time for graph connectivity via the Floyd-Warshall algorithm encoded in *CatSAT* takes approximately 40 microseconds on average for a graph of five vertices, and approximately 40 milliseconds on average for a graph of 20 vertices [12].

Graph connectivity is just one potential constraint which could be imposed on a randomly generated graph. Other constraints could prove to be even slower in runtime as pathfinding algorithms and graph search algorithms are nearly always supralinear. Letting V be the number of vertices in a graph and E the number of edegs, Dijkstra's algorithm has a time complexity of $O(E + V \log V)$, Bellman-Ford has a time complexity of $O(VE)$, Johnson's algorithm has a time complexity of $O(VE + V^2 \log V)$, and Floyd-Warshall has a time complexity of $O(V^3)$.

1.5 Paper Organization

Section 1 discussed the motivations behind this project, previous related work done, and challenges posed to the subject matter.

A more in-depth overview of SLS SAT solving will be presented in Section 2, along with more details regarding pseudocode and strengths and weaknesses of SLS SAT solvers.

Then, in Section 3, the codebase which was implemented will be described, beginning with an overview of how to use *CatSAT*, the data structures present to allow for the construction of graph constraints, and the implementations of the graph constraints themselves. It will also include a brief note on interpreting solution results, numerous examples with code snippets and figures, and conclude with runtime analysis.

Next, Section 4 will introduce the vast possibilities of applications of graph constraints in video games. In particular, a video game developed in Unity, *Why Can't We All Just Get Along?*, will be presented and detailed with code snippets and screenshots.

Finally, in Section 5, a concluding discussion and opportunities for future work will be given.

2 SLS SAT Solving

An introduction to SAT solvers was presented in Section 1.2. This section will explore generic SLS solving, the `WalkSAT` algorithm, arbitrary constraints, and the way these concepts will intersect in the work discussed in Section 3.

A generic SLS solver is given by the following pseudocode:

```
While not all constraints satisfied:
    Pick a random constraint
    Make a greedy choice for what
        variable to flip from the
        constraint
    Flip the chosen variable and update
         everything
```

The greediest choice of the variable to flip results in the largest decrease in the total number of unsatisfied clauses. Some constraint-based problems may be contradictory and hence will not have a solution, so it is very common to implement a maximum number of flips for the variables in a given constraint as well as a maximum number of attempts to solve the problem. This would make the pseudocode look more like this:

```
For i = 1 to MAX_TRIES:
    Pick a random constraint
    For j = 1 to MAX_FLIPS:
        Make a greedy choice for what
            variable to flip from the
            constraint
        Flip the chosen variable and
            update everything
If a solution was found, return it
Otherwise return an exception
```

Basic SAT solvers typically have only one kind of constraint, but they can be generalized to allow for arbitrary constraints as follows:

```
While not all constraints satisfied:
    Pick an unsatisfied constraint c
    Let v_c = the variables in c
    Let p_c ⊆ v_c = the variables in
        the constraint that would move
        c closer to being satisfied
    Let best = min_{v in p_c} cost(c)
    Flip best
    Update everything
```

For clauses, `p_c` is the same as `v_c`. For cardinality constraints, `p_c` is the set of false literals if the constraint is unsatisfied due to having too few true literals, or `p_c` is the set of true literals if the constraint is unsatisfied due to having too many true literals.

Each constraint has a `cost` function associated with it, which returns a number corresponding to how favorable or unfavorable it would be to flip a given variable within the constraint.

As mentioned previously, the issue with always choosing the variable to flip greedily is the possibility of the solver reaching a local minimum and getting stuck flipping the same variables over and over again. This is combated in the `WalkSAT` algorithm by introducing a noise parameter which controls whether the variable to flip is chosen randomly or greedily. The `WalkSAT` pseudocode is given by:

```
While not all constraints satisfied:
    Pick an unsatisfied constraint c,
        randomly
    Let v_c = the variables in c
    With probability p:
        Let rand_v = random variable in
             v_c
        Flip rand_v
        Update everything
    Else:
        Let p_c ⊆ v_c = the variables
            in the constraint that
            would move c closer to
```

```
            being satisfied
10          Let best_v = min_{v in p_c} cost(c)
11          Flip best_v
12          Update everything
```

Again, `WalkSAT` is often implemented with a maximum number of solution attempts and a maximum number of flips to ensure that the solver terminates in a reasonable amount of time.

The basic algorithm for making a connected graph is to use a spanning forest and greedily add edges that connect trees within the forest, until the forest has one overall connected component. Hence, this algorithm can be integrated into the above framework by making it a constraint (that the generated graph must be connected). The variables to be flipped are the edges in the graph, which are added to or removed from the graph when variables are flipped by the solver. This is a versatile approach to adding non-SAT greedy algorithms to SLS SAT solvers, and implementations of such are given in Section 3.

3 Graph Constraint Algorithms in *CatSAT*

3.1 General Idea

Using *CatSAT*, a constraint problem is initialized using

```
Problem p = new Problem();
```

Then, a graph is initialized for the problem to be solved, passing the problem `p`, the number of nodes, and optionally the initial density of the graph. For example, one may write

```
Graph graph = new Graph(p, 20, 0);
```

to initialize a graph with `20` nodes that has no edges to start. Note that the last field is optional and by default gives the new graph an initial density of `0.5`. From here, the user can add constraints to be imposed on the graph via their public accessors. For example, if the user desired for the generated graph to be connected, they would write

```
graph.AssertConnected();
```

If there is not a public accessor for a particular graph constraint or the user writes their own constraint, they can alternatively be imposed on the constraint problem to be solved by calling the `AddCustomConstraint` function. For example, if a user wrote a new graph constraint called `MyGraphConstraint` that takes the graph itself as input, they would write

```
p.AddCustomConstraint(new
    MyGraphConstraint(graph));
```

Once all desired graph constraints have been added, all that is left to do is to call the function that solves the problem:

```
p.Solve();
```

Note that *CatSAT* does have a maximum number of flips it will attempt until a timeout, which is available to the user to modify.

At the time of writing this paper, the list of constraints that a user can readily use are:

- `GraphConnectedConstraint`, called via the public accessor `AssertConnected`, which ensures that the graph generated is connected
- `SubsetConnectedConstraint`, called via the public accessor `AssertConnected`, which ensures that a subset of vertices in the graph generated are connected
- `NodesConnectedConstraint`, called via the public accessor `AssertNodesConnected`, which ensures that two specified nodes in the graph have a path between them
- `Density`, which is a public method that ensures that the density of the graph is contained between the specified lower and upper bounds
- `VertexDegree`, which is a public method that ensures a specified vertex has degree contained between the specified lower and upper bounds
- `AssertNBridges`, which is a public method that ensures that the number of edges between two specified subgraphs is contained between the specified lower and upper bounds

3.2 Data Structures

3.2.1 EdgeProposition

An `EdgeProposition` is a special type of proposition representing an edge between a specified `SourceVertex` and `DestinationVertex`. As a proposition is a type of literal, an `EdgeProposition` has a truth value that gets flipped by the boolean SAT solver. An `EdgeProposition` (n, m) with a truth value of `true` indicates that the edge between vertices n and m is present in the graph, and a truth value of `false` indicates that the edge between vertices n and m is not present in the graph. Thus, the "flipping" of an `EdgeProposition`'s truth value encompasses adding the edge (n, m) to the graph if the truth value flips from `false` to `true`, and removing the edge (n, m) from the graph if the truth value flips from `true` to `false`.

3.2.2 Graph

A `Graph` is the data structure representing the problem *CatSAT* is attempting to solve. It is constructed with a `Problem`, an integer number of vertices in the graph, and optionally an initial density to the graph. When the constructor is called, an integer array is populated with the vertices of the graph, a spanning forest is initialized, an empty list of `Subgraphs` is initialized, and two dictionaries are built that map every possible `EdgeProposition` (n, m) to its corresponding `SATVariable` index, and vice versa.

As `EdgePropositions` are flipped while the boolean SAT solver is running, the `Graph`'s `SpanningForest` is maintained and updated accordingly with functions `ConnectInSpanningForest` (when adding an edge) and `Disconnect` (when removing an edge).

`ConnectInSpanningForest` adds the edge (n, m) to the spanning forest by calling `Union`:

```
1 public void ConnectInSpanningForest(
      int n, int m)
2 {
3     bool edgeAdded = SpanningForest.
          Union(n, m);
4     if (edgeAdded) Console.WriteLine($"
          Connected {n} and {m} in Graph
          .");
5 }
```

See Section 3.2.4 for more information regarding the `SpanningForest` `Union` function.

If the edge is present in `SpanningForest`, `Disconnect` clears the graph's spanning forest and rebuilds it without the edge that was specified to be removed:

```
1 public void Disconnect(int n, int m)
2 {
3     ushort edgeIndex = Edges(n, m).
          Index;
4     if (!SpanningForest.Contains(
          edgeIndex)) return;
5
6     SpanningForest.Clear();
7     _spanningForestBuilt = false;
8     Console.WriteLine($"Disconnected {n
          } and {m} in Graph.");
9     RebuildSpanningForest();
10 }
```

Rebuilding a spanning forest constitutes calling `ConnectInSpanningForest` for every edge proposition that has truth value set to `true` in the boolean solver's current solution state:

```
1 private void RebuildSpanningForest()
2 {
3     SpanningForest.Clear();
4     IEnumerable<EdgeProposition>
          trueEdges = SATVariableToEdge.
          Values.Where(edge => Solver.
          Propositions[edge.Index]);
```

```
    foreach (EdgeProposition
        edgeProposition in trueEdges)
    {
        ConnectInSpanningForest(
            edgeProposition.
            SourceVertex,
            edgeProposition.
            DestinationVertex);
    }

    _spanningForestBuilt = true;
}
```

3.2.3 Subgraph

A `Subgraph` is a data structure representing a subset of a `Graph`. It is constructed with a reference to its original `Graph` and a list of the integer vertices present in the subgraph. When the constructor is called, the vertices are copied into a local integer array and the `SATVariableToEdge` dictionary is populated with edge propositions in the original graph that occur exclusively between vertices in the subgraph. Then, a spanning forest is initialized and the subgraph is added to the original graph's list of subgraphs.

`EdgePropositions` in a `Subgraph` are flipped in the same way that they are flipped in a `Graph`, as `Subgraph` has its own analogous definitions of the functions `ConnectInSpanningForest`, `Disconnect`, and `RebuildSpanningForest`.

3.2.4 SpanningForest

A `SpanningForest` is a modified Union-Find data structure used by `Graphs` and `Subgraphs` to keep track of connected components and paths relevant to the various constraints that may be imposed on a given SAT problem. It is initialized either with a `Graph` or a `Subgraph`. When the constructor is called, the number of connected components is set to the number of vertices in the `Graph` or `Subgraph`, an integer array of representatives and ranks is initialized, and a list of edges in the spanning forest is initialized. Initially, each vertex is its own representative, and its rank is `0`.

The representative of a vertex is found by recursively calling the `Find` function:

```
private int Find(int n)
{
    int nRep = _repsAndRanks[n].
        representative;
    return nRep == n ? n : Find(nRep);
}
```

Two vertices are joined in a spanning forest by calling the `Union` function, which sets the representatives of the two vertices to be the same, choosing the representative by whichever vertex has higher rank. If the vertices did not already have the same representative, the number of connected components is decremented and the function returns `true`. The function is given below:

```
public bool Union(int n, int m)
{
    int nRep = Find(n);
    int mRep = Find(m);

    if (nRep == mRep) return false;

    ushort edge = _graph.
        EdgeToSATVariable[_graph.Edges(
        n, m)];
    _edges.Add(edge);

    int nRank = _repsAndRanks[nRep].
        rank;
    int mRank = _repsAndRanks[mRep].
        rank;
    if (nRank < mRank)
    {
        _repsAndRanks[nRep].
            representative = mRep;
    }
    else if (nRank > mRank)
    {
```

```
        _repsAndRanks[mRep].
            representative = nRep;
    }
    else
    {
        _repsAndRanks[mRep].
            representative = nRep;
        _repsAndRanks[nRep].rank++;
    }

    ConnectedComponentCount--;
    return true;
}
```

Two vertices are in the same equivalence class if they have the same representative:

```
public bool SameClass(int n, int m) =>
    Find(n) == Find(m);
```

SpanningForests also have functions to determine if the addition of an edge would result in connecting two vertices and if the removal of an edge would result in the disconnecting of two vertices. The addition of an edge results in the vertices n and m being connected if either:

- n has the same representative as the added edge's source vertex, and m has the same representative as the added edge's destination vertex, or
- n has the same representative as the added edge's destination vertex, and m has the same representative as the added edge's source vertex.

```
public bool WouldConnect(int n, int m,
    EdgeProposition edge)
{
    int nRep = Find(n);
    int mRep = Find(m);
    int sourceRep = Find(edge.
        SourceVertex);
    int destRep = Find(edge.
        DestinationVertex);
    return (nRep == sourceRep && mRep
        == destRep) || (nRep == destRep
         && mRep == sourceRep);
}
```

The removal of an edge may disconnect two vertices in the same equivalence class if the edge being removed is present in the spanning forest:

```
public bool MightDisconnect(
    EdgeProposition edge) => _edges.
    Contains(edge.Index);
```

3.2.5 CustomConstraint

A CustomConstraint is a subclass of Constraint in *CatSAT* that allows developers to create a constraint with additional custom fields and properties.

3.3 Algorithms

In general, each algorithm utilizes a cost function to determine whether a particular edge should be flipped (which, in this context, means to say the edge is added if it is not currently present or removed if it is currently present) by the SAT solver given the current state of the problem. There are two particular cost functions: AddingRisk used to assess the favorability of adding a particular edge to the graph, and RemovingRisk used to assess the favorability of removing a particular edge from the graph. Furthermore, each algorithm internally maintains a spanning forest which is crucial in evaluating the cost of flipping an edge in the graph.

3.3.1 GraphConnectedConstraint

In order for a graph to be connected, it must have one connected component. An edge is then considered to be flipped based on the following logic:

```
public override int CustomFlipRisk(
    ushort index, bool adding)
{
    int componentCount = SpanningForest
        .ConnectedComponentCount;
    if (componentCount == 1 && adding)
        return 0;
    EdgeProposition edge = Graph.
        SATVariableToEdge[index];
    return adding ? AddingRisk(edge) :
        RemovingRisk(edge);
}
```

Then, the cost associated with adding an edge is given by the following logic:

```
private int AddingRisk(EdgeProposition
     edge) => Graph.AreConnected(edge.
    SourceVertex, edge.
    DestinationVertex) ? 0 :
    EdgeAdditionRisk;
```

The cost associated with removing an edge is given by the following logic:

```
private int RemovingRisk(
    EdgeProposition edge) =>
    SpanningForest.Contains(edge.Index
    ) ? EdgeRemovalRisk : 0;
```

These three functions mean to evaluate the following:

- Adding an edge is favorable if it results in the source and destination vertices being in the same equivalence class, and adding an edge is never unfavorable.
- Removing an edge is unfavorable if it is currently in the spanning forest being maintained for the graph, and removing an edge is never favorable.
- If there is exactly one connected component in the graph (meaning that the graph itself is connected), then adding an edge is neither favorable nor unfavorable.

The `GreedyFlip` function in *CatSAT* is overridden in `GraphConnectedConstraint` to favor flipping edges that are between vertices which are not in the same equivalence class.

The `GraphConnectedConstraint` is satisfied when the spanning forest of the `Graph` has a `ConnectedComponentCount` of `1`.

3.3.2 `SubsetConnectedConstraint`

The `SubsetConnectedConstraint` behaves in a similar nature to the `GraphConnectedConstraint`, except it operates on a `Subgraph`, or a subset of the vertices of the original `Graph`. A `Subgraph` maintains its own spanning forest separate from that of the original `Graph`, updating both itself and the original `Graph`'s spanning forest when edges are added and removed.

Note that a `Subgraph` is initialized with the `Graph` from which it originates and an `IEnumerable` of the vertices it contains. For example:

```
Problem p = new Problem();
Graph g = new Graph(p, 5, 0);
Subgraph s = new Subgraph(g, new[] {
    0, 1, 2 });
```

This code creates a `Graph` with five vertices (labeled `0`, `1`, `2`, `3`, and `4`) and a `Subgraph` of `graph` containing the vertices `0`, `1`, and `2`.

The logic for `CustomFlipRisk`, `AddingRisk`, `RemovingRisk`, and `GreedyFlip` are otherwise the same as for those in `GraphConnectedConstraint`.

The `SubsetConnectedConstraint` is satisfied when the spanning forest of the `Subgraph` has a `ConnectedComponentCount` of `1`.

3.3.3 NodesConnectedConstraint

Two nodes, namely `SourceNode` and `DestinationNode`, in a graph are considered connected if there exists a path between them. This constraint in particular has two primary modes of operation based on whether the path between the nodes has been constructed while a model has not yet been found.

If the path has not yet been constructed, an edge is considered to be flipped based on the following logic:

```
public override int CustomFlipRisk(ushort index, bool adding)
{
    EdgeProposition edge = Graph.SATVariableToEdge[index];
    bool previouslyConnected = Graph.AreConnected(edge.SourceVertex, edge.DestinationVertex);
    if (previouslyConnected && adding)
        return 0;
    return adding ? AddingRisk(edge) : RemovingRisk(edge);
}
```

Then, the cost associated with adding an edge is given by the following logic:

```
private int AddingRisk(EdgeProposition edge)
{
    if (SpanningForest.WouldConnect(SourceNode, DestinationNode, edge)) return EdgeAdditionRisk * 2;
    return Graph.AreConnected(edge.SourceVertex, edge.DestinationVertex) ? 0 : EdgeAdditionRisk;
}
```

The cost associated with removing an edge is given by the following logic:

```
private int RemovingRisk(EdgeProposition edge)
{
    return SpanningForest.Contains(edge.Index) ? EdgeRemovalRisk : 0;
}
```

These three functions mean to evaluate the following:

- Adding an edge is extremely favorable if it results in `SourceNode` and `DestinationNode` being in the same equivalence class. Adding an edge is favorable if the edge's source and destination vertices are not currently in the same equivalence class, and adding an edge is never unfavorable.
- Removing an edge is unfavorable if it is currently in the spanning forest being maintained for the graph, and removing an edge is never favorable.
- If the edge's source and destination vertices were already connected (in the same equivalence class) prior to the flip, then adding an edge is neither favorable nor unfavorable.

Alternatively, if the path has been constructed and there are still constraints which are yet to be satisfied, then the boolean flag keeping track of this state is set to true and a breadth-first search runs (via the `ShortestPath` method) to find the shortest path between `SourceNode` and `DestinationNode`, storing these edges in a list of their indices. Then, the logic that runs for `CustomFlipRisk` and `AddingRisk` is the same as previously stated. However, the cost associated with removing an edge differs, instead given by the following logic:

```
private int RemovingRisk(EdgeProposition edge)
{
    return _edgesInPath.Contains(edge.Index) ? EdgeRemovalRisk * 2 : 0;
}
```

This means to say that it is extremely unfavorable to remove an edge that is in the path connecting `SourceNode` and `DestinationNode`, otherwise it is neither favorable nor unfavorable.

The logic for `GreedyFlip` is the same as that in `GraphConnectedConstraint` and `SubsetConnectedConstraint`.

The `NodesConnectedConstraint` is satisfied when `SourceNode` and `DestinationNode` are connected via some path and hence are in the same equivalence class.

3.3.4 `Density`

A graph's density is the ratio of edges present in a graph to the total number of possible edges in the graph. The `Density` constraint asserts that a graph has density between a specified minimum and maximum (inclusive of both bounds). This constraint utilizes *CatSAT*'s `Quantify` constraint, which asserts that the number of true literals in a specified set is bounded by a specified minimum and maximum (also inclusive of both bounds). The logic is given by:

```
1 public void Density(float min, float
      max)
2 {
3     int edgeCount = SATVariableToEdge.
          Count;
4     IEnumerable<EdgeProposition> edges
          = SATVariableToEdge.Values;
5     int minEdges = (int)Math.Round(min
          * edgeCount);
6     int maxEdges = (int)Math.Round(max
          * edgeCount);
7     Problem.Quantify(minEdges, maxEdges
          , edges);
8 }
```

Note that both `Graph` and `Subgraph` have a `Density` constraint.

3.3.5 `VertexDegree`

The `VertexDegree` constraint asserts that a specified vertex has degree between specified minimum and maximum values (inclusive of both bounds). This constraint also utilizes `Quantify`, and the logic is given by:

```
1 public void VertexDegree(int vertex,
      int min, int max)
2 {
3     IEnumerable<EdgeProposition>
          incidentEdges = from v in
          Vertices where v != vertex
          select Edges(v, vertex);
4     Problem.Quantify(min, max,
          incidentEdges);
5 }
```

Note that both `Graph` and `Subgraph` have a `VertexDegree` constraint.

3.3.6 `AssertNBridges`

A bridge is an edge whose removal would result in an increase in the number of connected components in a graph. The `Graph` class has a public constraint method `AssertNBridges` that ensures that the number of such bridge edges between two specified `Subgraphs` is contained between a minimum and maximum number (inclusive). It also uses the `Quantify` method to impose a constraint on the bridge edges that could exist between two `Subgraphs`. These bridges are found with the following logic:

```
1 private IEnumerable<EdgeProposition>
      EdgesBetweenSubgraphs(Subgraph s1,
       Subgraph s2)
2 {
3     List<EdgeProposition> edges = (from
          v1 in s1.Vertices from v2 in
          s2.Vertices select Edges(v1, v2
          )).ToList();
4     return edges;
5 }
```

Then, `AssertNBridges` is given by:

```
public void AssertNBridges(int min,
    int max, Subgraph s1, Subgraph s2)
{
    IEnumerable<EdgeProposition>
        bridges = EdgesBetweenSubgraphs
        (s1, s2);
    Problem.Quantify(min, max, bridges)
        ;
}
```

3.4 Interpreting Results

Calling the `Solve` function on an initialized `Problem` in *CatSAT* will not inherently display the generated graph. However, it is fairly simple to obtain relevant properties of the `Solution`: the `Graph`'s vertices, all possible `EdgePropositions`, and the list of `EdgePropositions` which have a truth assignment of `true` in the outputted model. Suppose we instantiate and solve the following problem:

```
public void InterpretingResults()
{
    Problem p = new Problem();
    Graph g = new Graph(p, 10);
    g.AssertNodesConnected(0, 1);
    g.AssertConnected();
    Solution s = p.Solve();
}
```

Both before and after `Solve()` is executed, the `Graph`'s vertices can be retrieved using

```
int[] vertices = g.Vertices;
```

Similarly, all possible `EdgePropositions` in the `Graph` can be obtained by writing

```
IEnumerable<EdgeProposition> allEdges
    = g.SATVariableToEdge.Values;
```

Finally, after `Solve()` is executed, the `EdgePropositions` set to true (equivalently, the edges which are present in the model after solving has concluded) acquired with

```
IEnumerable<EdgeProposition> trueEdges
    = SATVariableToEdge.Select(pair
    => pair.Value).Where(edge => s[
    edge]);
```

3.5 Examples

In this section, multiple examples of code snippets and corresponding graphs generated by *CatSAT* are provided for the various constraints described in Section 3.3. Note that, in the following figures, green edges represent those which are present in the `Graph`'s `SpanningForest`, and red edges represent those which are not present.

First, consider a simple example of a `GraphConnectedConstraint` imposed on a `Graph` with five vertices and an initial edge probability of `0`. With such an initial probability, a tree will be generated, as the algorithm will keep adding edges that reduce the number of connected components until the graph reaches one connected component.

```
public void FigureOne()
{
    Problem p1 = new Problem();
    Graph g1 = new Graph(p1, 5, 0);
    g1.AssertConnected();
    p1.Solve();
}
```

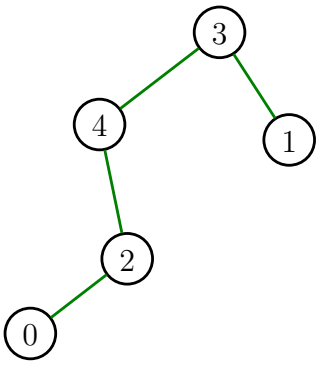

Figure 1: Connected graph with five vertices and initial edge probability of `0`.

Note that, if the edge probability is not specified, it is by default set to `0.5f`, and there may be additional edges present in the solution graph that are not in the spanning forest. As such, the initial edge probability essentially functions as a soft density constraint. In the absence of a `Density` constraint, an initial edge probability of `p` will likely result in a generated graph whose density is very close to `p`.

```
1 public void FigureTwo()
2 {
3     Problem p2 = new Problem();
4     Graph g2 = new Graph(p2, 5);
5     g2.AssertConnected();
6     p2.Solve();
7 }
```

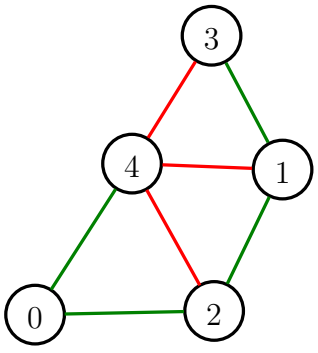

Figure 2: Connected graph with five vertices and initial edge probability of `0.5f`.

The `GraphConnectedConstraint` also works very quickly for graphs of even larger size.

```
1 public void FigureThree()
2 {
3     Problem p3 = new Problem();
4     Graph g3 = new Graph(p3, 40, 0);
5     g3.AssertConnected();
6     p3.Solve();
7 }
```

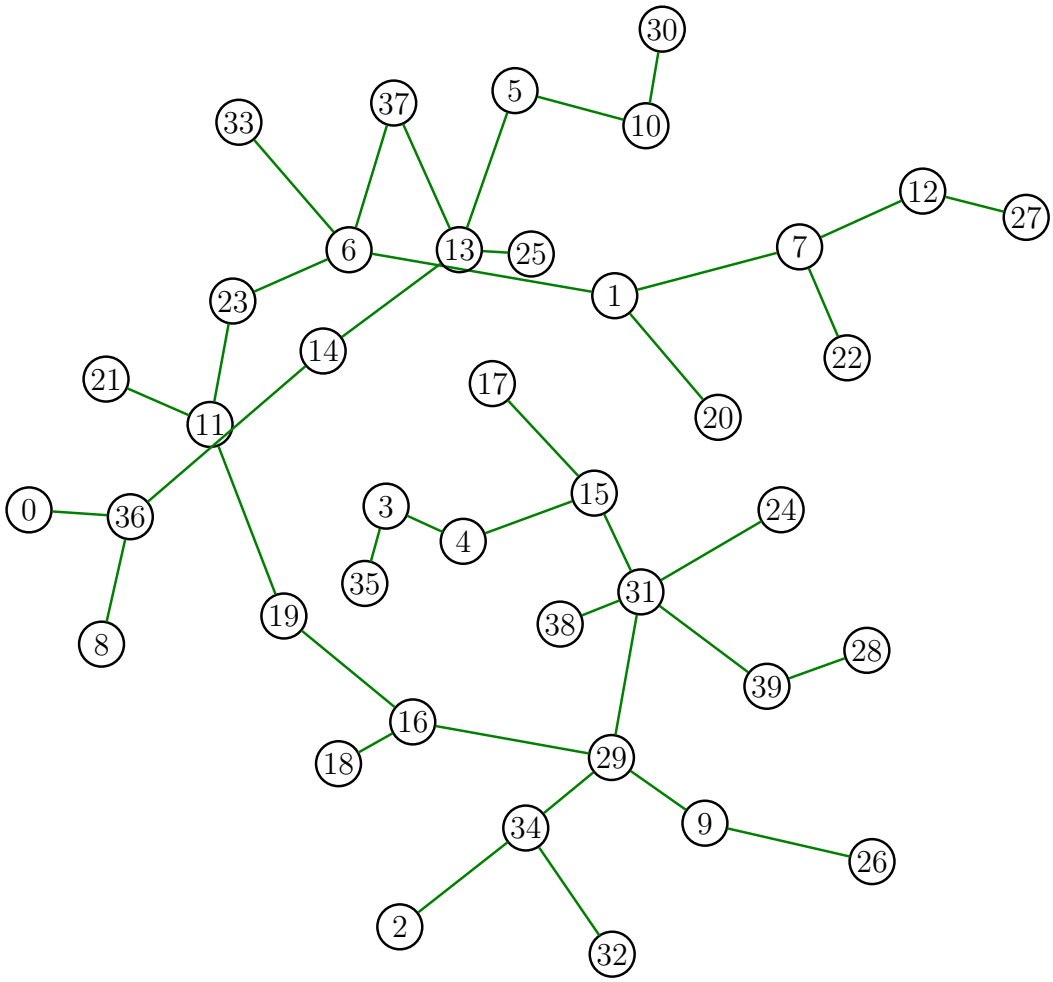

Figure 3: Connected graph with 40 vertices and initial edge probability of `0`.

When generating graphs with an even larger number of vertices, note that having an initial edge probability greater than `0` aids with solution time and increases the likelihood that *CatSAT* does not time out. This is exemplified when generating a graph with, say, 100 vertices.

```
1 public void FigureFour()
2 {
3     Problem p4 = new Problem();
4     Graph g4 = new Graph(p, 100);
5     g4.AssertConnected();
6     p4.Solve();
7 }
```

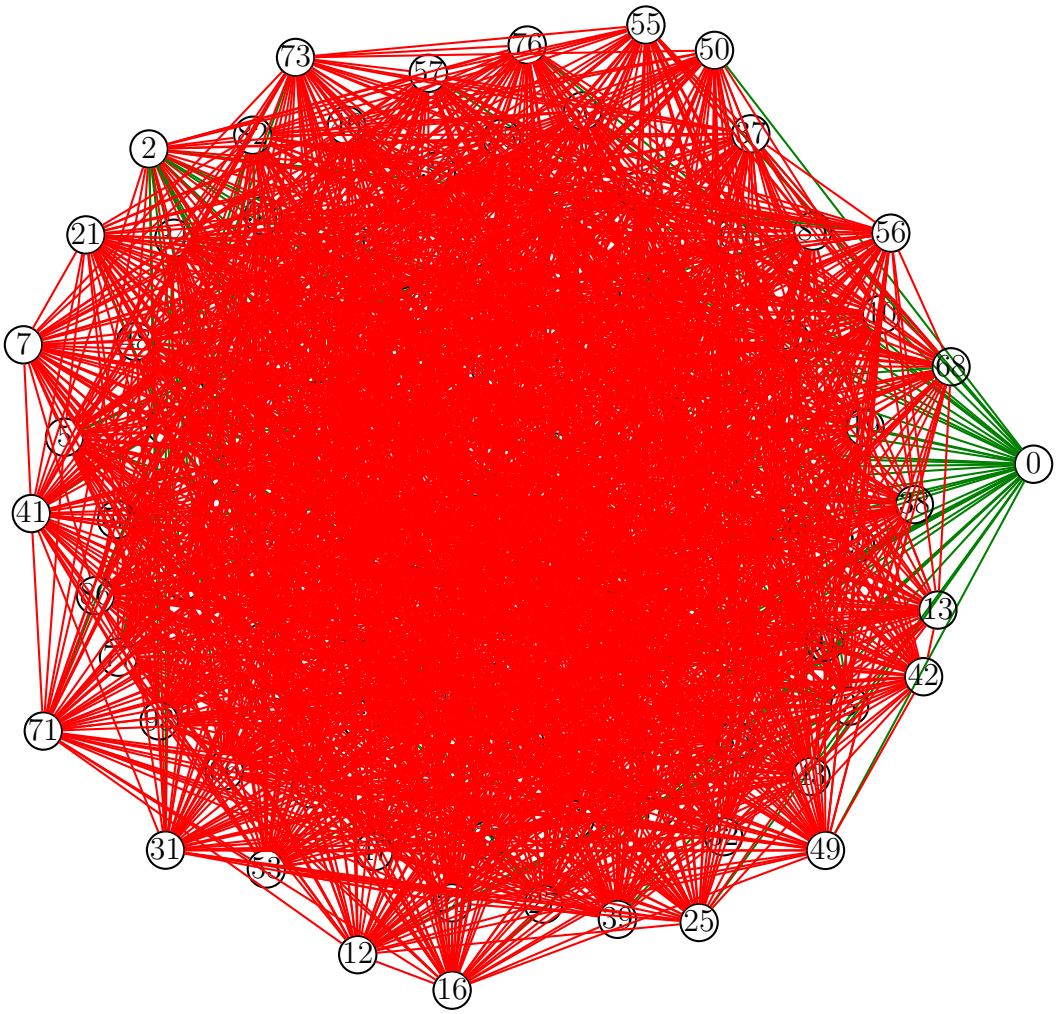

Figure 4: Connected graph with 100 vertices and initial edge probability of `0.5f`.

Now consider generating a graph that has a path between two specified vertices, for example, nodes `0` and `1`. Note that the path is shown in Figure 5 in blue, whereas other edges in the spanning tree are in green.

```
1 public void FigureFive()
2 {
3     Problem p5 = new Problem();
4     Graph g5 = new Graph(p5, 10, 0);
5     g5.AssertNodesConnected(0, 1);
6     p5.Solve();
7 }
```

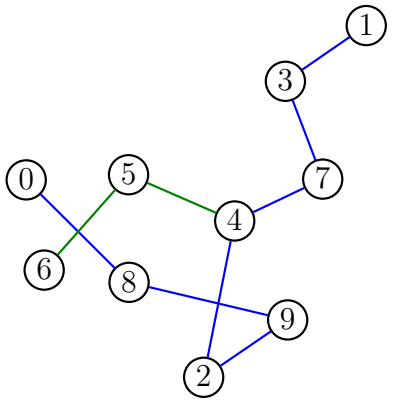

Figure 5: Path generated between vertices `0` and `1` (blue).

Multiple assertions that paths between different vertices in a graph exist, and again, increasing the likelihood that *Cat-*

SAT does not time out when finding a solution is achieved by increasing the initial edge probability. Note that, in Figure 6, a path between vertices `0` and `1` is shown in blue, a path between vertices `2` and `3` is shown in violet, edges in the spanning forest are shown in green, and edges not in the spanning forest are shown in red. In this example, the graph itself is also asserted to be connected.

```
public void FigureSix()
{
    Problem p6 = new Problem();
    Graph g6 = new Graph(p6, 20);
    g6.AssertNodesConnected(0, 1);
    g6.AssertNodesConnected(2, 3);
    g6.AssertConnected();
    g6.Solve();
}
```

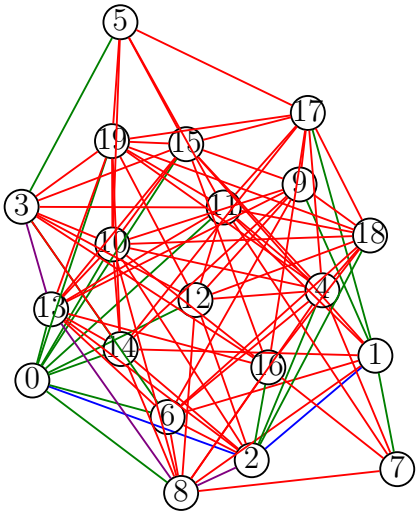

Figure 6: Connected graph with a path between vertices `0` and `1` (blue), and a path between vertices `2` and `3` (violet).

The `Density` constraint can be used in absence of the `GraphConnectedConstraint` to assert that a certain proportion of edges exist in the generated graph. In a graph with five vertices, there are 10 possible edges that can exist. In the following example, it is asserted that exactly two edges are present in the generated graph.

```
public void FigureSeven()
{
    Problem p7 = new Problem();
    Graph g7 = new Graph(p7, 5, 0);
    g7.Density(0.2f, 0.2f);
    p7.Solve();
}
```

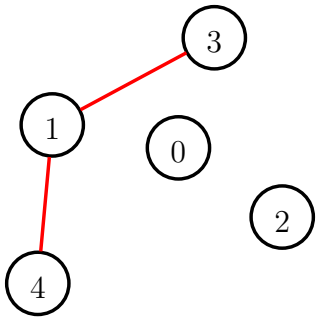

Figure 7: Graph with five vertices and 20% density (two edges).

For another example, nine edges can be asserted to be present. Note that, in the graph generated in Figure 8, all possible edges exist in the graph except the one from vertices `0` to `1`.

```
public void FigureEight()
{
    Problem p8 = new Problem();
    Graph g8 = new Graph(p8, 5, 0);
    g8.Density(0.9f, 0.9f);
    p8.Solve();
}
```

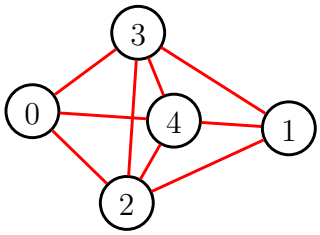

Figure 8: Graph with five vertices and 90% density (nine edges).

One can also generated connected graphs of specified density, and a solution will be found so long as the minimum specified density is at least that which is required to connect the graph. The minimum and maximum density values can be calculated from the number of vertices, the minimum desired degree of each vertex, and the maximum desired degree of each vertex with:

```
public (float, float) CalculateDensity(int numVertices, int minDegree, int maxDegree)
{
    int numEdges = numVertices * (numVertices - 1) / 2;
    int minEdges = numVertices * minDegree / 2;
    int maxEdges = numVertices * maxDegree / 2;
    return ((float) minEdges / numEdges, (float) maxEdges / numEdges);
}
```

Hence, for example, a connected graph can be generated such that it is connected, with each vertex having degree 1 or 2 with:

```
public void FigureNine()
{
    Problem p9 = new Problem();
    Graph g9 = new Graph(p9, 20);
    (float, float) densityBounds = CalculateDensity(20, 1, 2);
    g9.Density(densityBounds.Item1, densityBounds.Item2);
    g9.AssertConnected();
    p9.Solve();
}
```

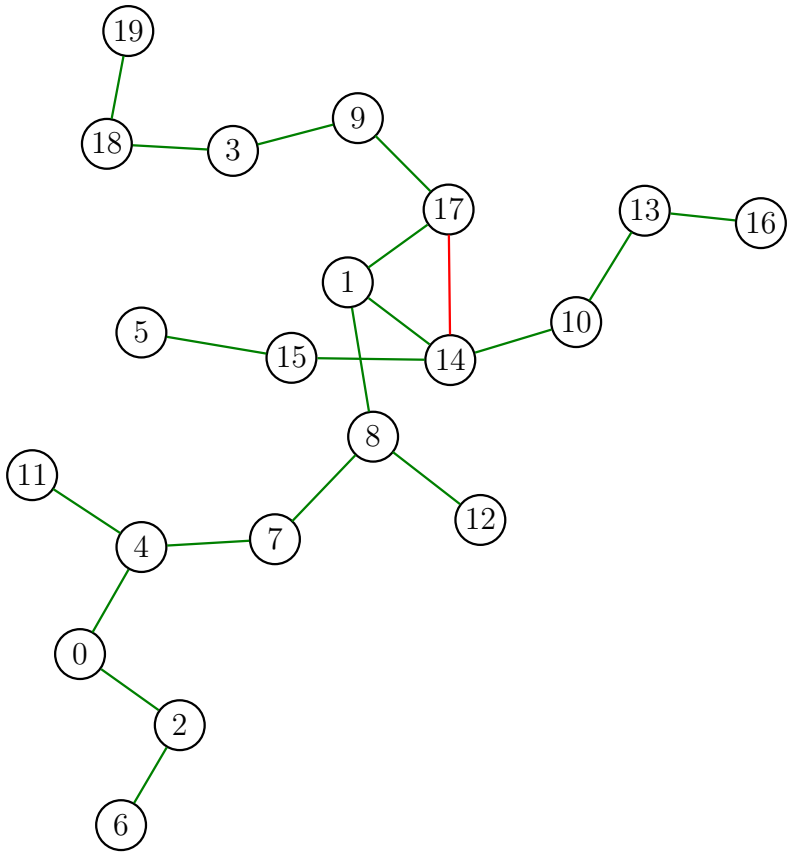

Figure 9: Connected graph with low density.

The `VertexDegree` constraint can be used in combination with the `GraphConnectedConstraint` to assert that the generated graph is a cycle. In a cycle, the graph must be connected with every vertex having a degree of 2.

```
public void FigureTen()
{
    Problem p10 = new Problem();
    Graph g10 = new Graph(p10, 10, 0);
    foreach (int v in g9.Vertices)
    {
        g9.VertexDegree(v, 2, 2);
    }
    g10.AssertConnected();
    p10.Solve();
}
```

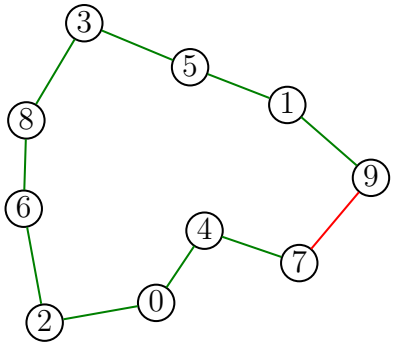

Figure 10: 10-cycle, given by a connected graph with 10 vertices and each vertex having degree 2.

A graph can also be generated with subgraphs that are connected. For example, a graph of 12 vertices can be split into two connected subgraphs of equal size (six vertices).

```
public void FigureEleven()
{
```

```
    Problem p11 = new Problem();
    Graph g11 = new Graph(p11, 12, 0);
    Subgraph s1 = new Subgraph(g11, new
        [] { 0, 1, 2, 3, 4, 5 });
    Subgraph s2 = new Subgraph(g11, new
        [] { 6, 7, 8, 9, 10, 11 });
    s1.AssertConnected();
    s2.AssertConnected();
    p11.Solve();
}
```

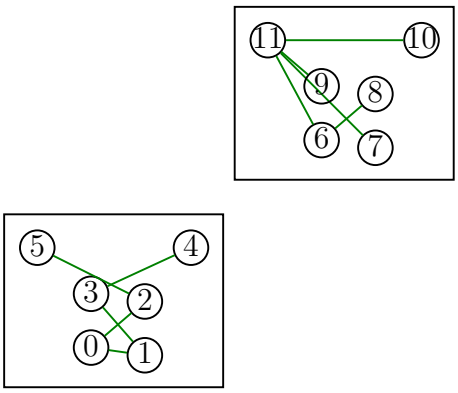

Figure 11: The subgraphs (0, 1, 2, 3, 4, 5) and (6, 7, 8, 9, 10, 11) are each connected in the graph.

The `AssertNBridges` constraint can be used to generate some number of edges between two provided subgraphs in a graph. Here, the bridges are shown in red, and the edges in the spanning forest of each subgraph are shown in green.

```
public void FigureTwelve()
{
    Problem p12 = new Problem();
    Graph g12 = new Graph(p12, 12, 0);
    Subgraph s3 = new Subgraph(g12, new
        [] { 0, 1, 2 });
    Subgraph s4 = new Subgraph(g12, new
        [] { 3, 4, 5 });
    s3.AssertConnected();
    s4.AssertConnected();
    g12.AssertNBridges(2, 2, s3, s4);
    p12.Solve();
}
```

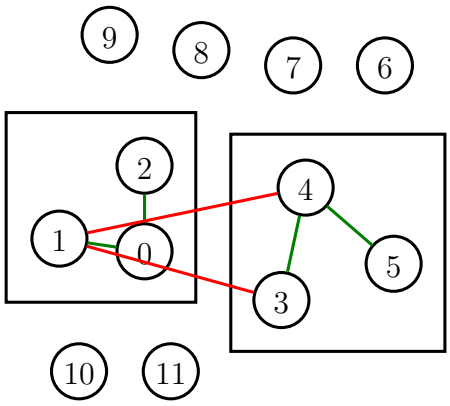

Figure 12: Two bridges generated between two subgraphs (red).

As mentioned shown earlier, multiple constraints can be applied to the same problem. For instance:

```
public void FigureThirteen()
{
    Problem p13 = new Problem();
    Graph g13 = new Graph(p13, 15);
    Subgraph s1 = new Subgraph(g13, new[] { 1, 2, 3, 4, 5 });
    Subgraph s2 = new Subgraph(g13, new[] { 10, 13 });
    Subgraph s3 = new Subgraph(g13, new[] { 12 });
    s1.AssertConnected();
    s2.AssertConnected();
    g13.Density(0.2f, 0.3f);
    g13.AssertNodesConnected(0, 10);
    g13.AssertNodesConnected(9, 14);
    g13.VertexDegree(12, 4, 5);
    g13.AssertConnected();
    p13.Solve();
}
```

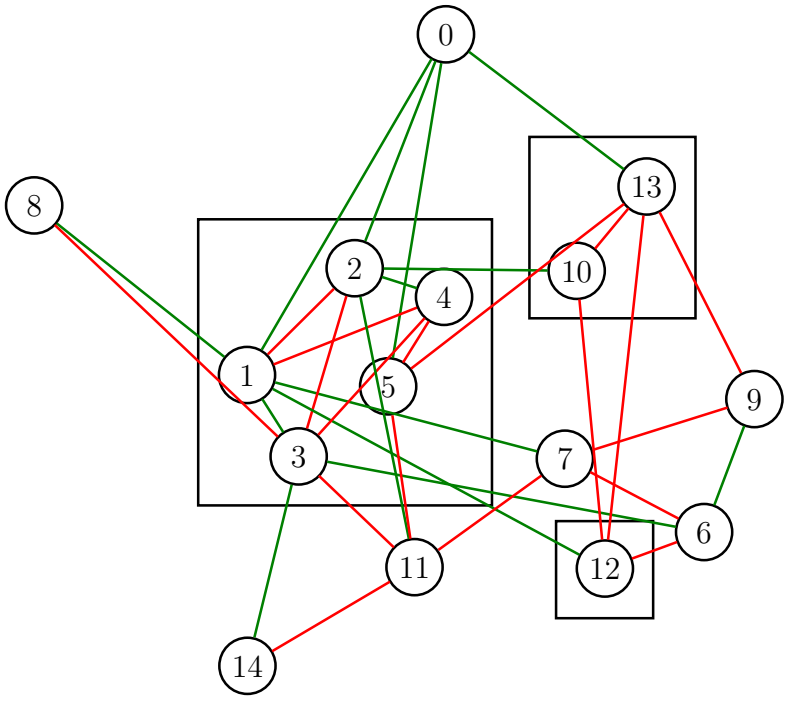

Figure 13: Multiple constraints imposed on one randomly generated graph.

3.6 Evaluation

In this section, various line plots showing performance of the system on several procedurally generated graph problems are displayed for different constraints. Tests were run single-threaded on a 2020 laptop with a 2.60 GHz Intel i7-10750H processor and 32GB RAM. First, the algorithms run are presented, with the plots following. Note that `size` is the current number of vertices in the graph, spanning the list { `5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100` }.

```
public void GraphConnected()
{
    Problem p = new Problem();
    Graph g = new Graph(p, size);
    g.AssertConnected();
    p.Solve();
}
```

```
public void NodesConnected()
{
    Problem p = new Problem();
    Graph g = new Graph(p, size);
    g.AssertNodesConnected(0, 1);
    p.Solve();
}
```

```
public void TwoSubgraphsConnected()
{
    Problem p = new Problem();
    Graph g = new Graph(p, size);
    int[] s1Vertices = Enumerable.Range
        (0, size / 2).ToArray();
    int[] s2Vertices = Enumerable.Range
        (size / 2, size - s1Vertices.
        Length).ToArray();
    Subgraph s1 = new Subgraph(g,
        s1Vertices);
    Subgraph s2 = new Subgraph(g,
        s2Vertices);
    s1.AssertConnected();
    s2.AssertConnected();
    p.Solve();
}
```

```
public void Cycle()
{
    Problem p = new Problem();
    Graph g = new Graph(p, size);
    foreach (int v in g.Vertices)
    {
        g.VertexDegree(v, 2, 2);
    }
    p.Solve();
}
```

```
public void Density()
{
    Problem p = new Problem();
    Graph g = new Graph(p, size);
    g.Density(0.5f, 0.5f);
    p.Solve();
}
```

```
public void Bridges()
{
    Problem p = new Problem();
    Graph g = new Graph(p, size);
    int[] s1Vertices = Enumerable.Range
        (0, size / 2).ToArray();
    int[] s2Vertices = Enumerable.Range
        (size / 2, size - s1Vertices.
        Length).ToArray();
    Subgraph s1 = new Subgraph(g,
        s1Vertices);
    Subgraph s2 = new Subgraph(g,
        s2Vertices);
    g.AssertNBridges(size / 2, size /
        2, s1, s2);
    p.Solve();
}
```

In each method defined above, after the `Problem`, `Graph`, and any additional components needed for constraints were initialized, solely the `p.Solve()` function was called 100 times, and the following statistics were collected, removing outliers:

- average runtime (milliseconds)
- median runtime (milliseconds)
- standard deviation in runtime (milliseconds)
- minimum runtime (milliseconds)
- maximum runtime (milliseconds)

Note that the error bars in the following plots represent one standard deviation above and below the average.

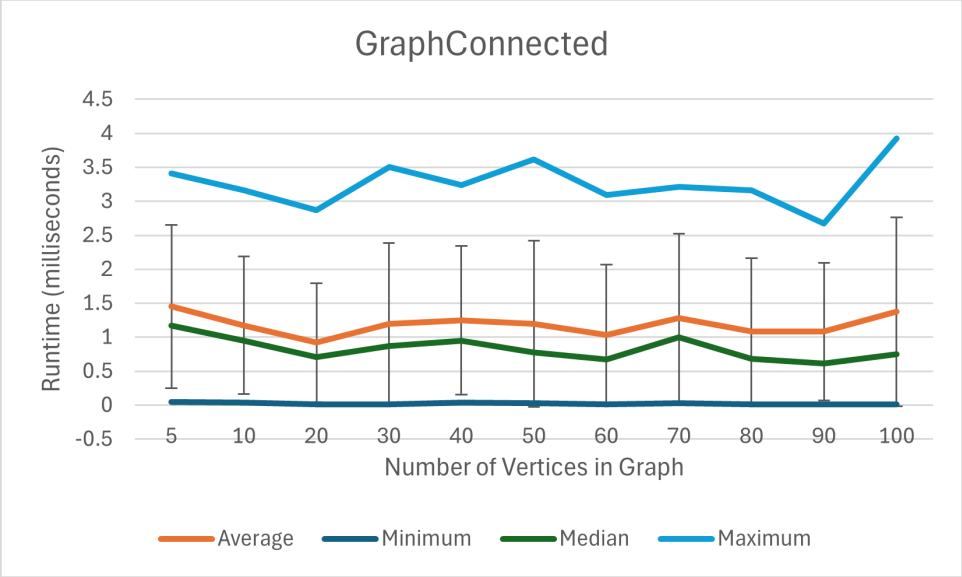

Figure 14: Runtime for `GraphConnectedConstraint`.

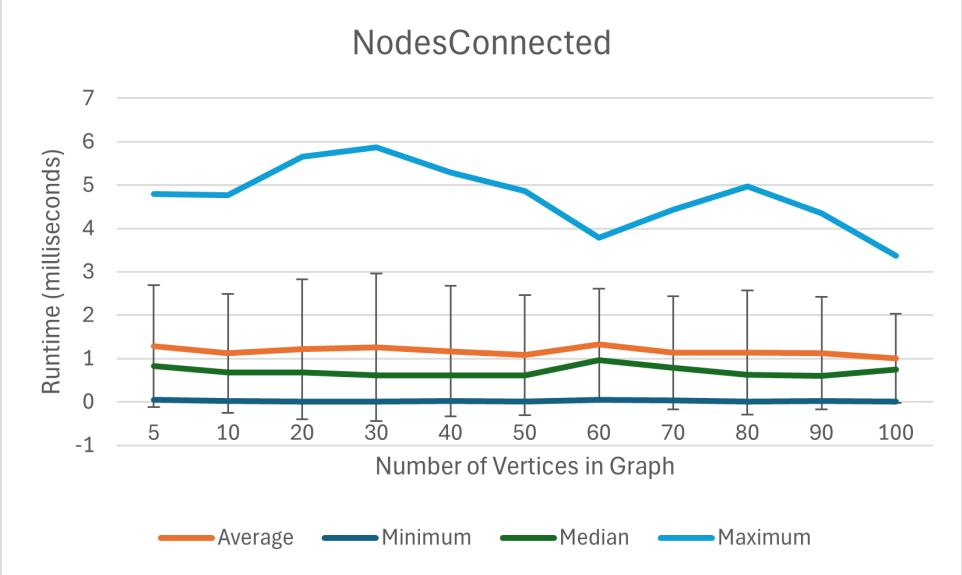

Figure 15: Runtime for `NodesConnectedConstraint` between vertices `0` and `1`.

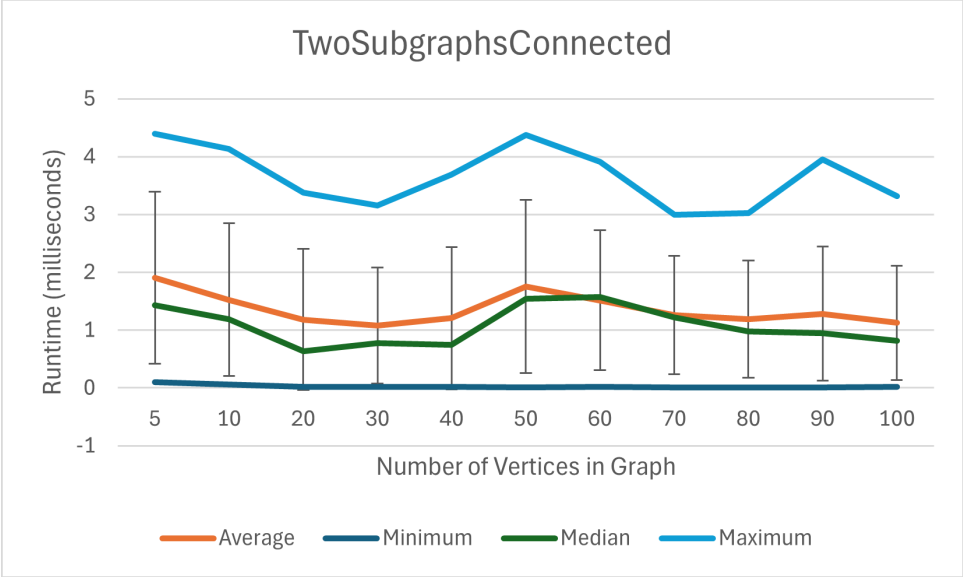

Figure 16: Runtime for `SubsetConnectedConstraint` on two `Subgraphs` of equal size.

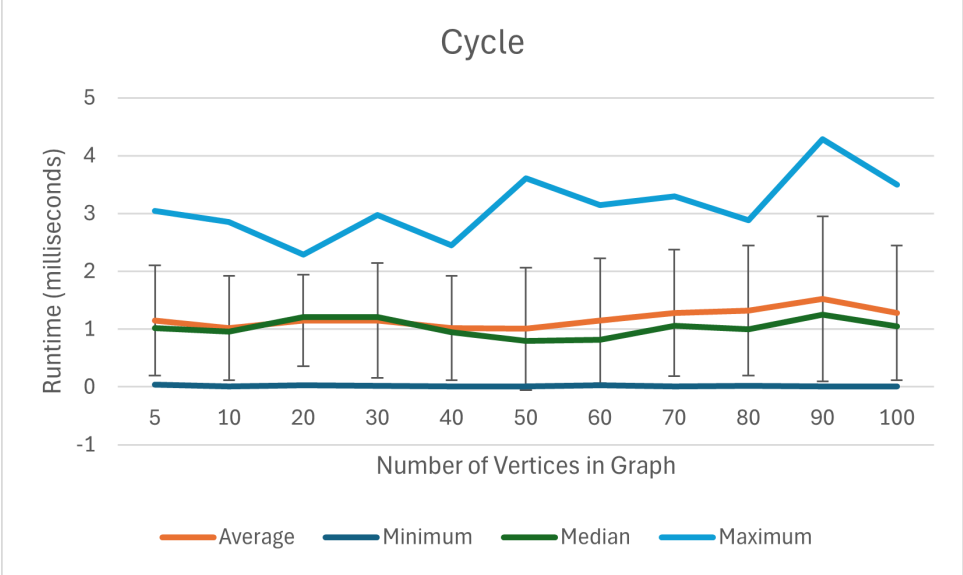

Figure 17: Runtime for the `VertexDegree` constraint such that each vertex has degree `2`, and a `GraphConnectedConstraint`.

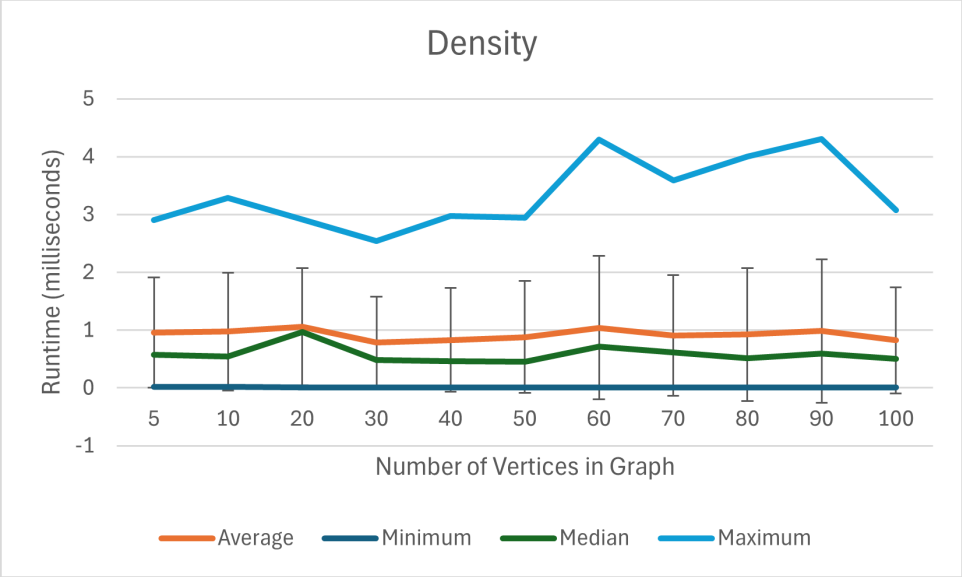

Figure 18: Runtime for the `Density` constraint such that the `Graph` has half of all possible edges.

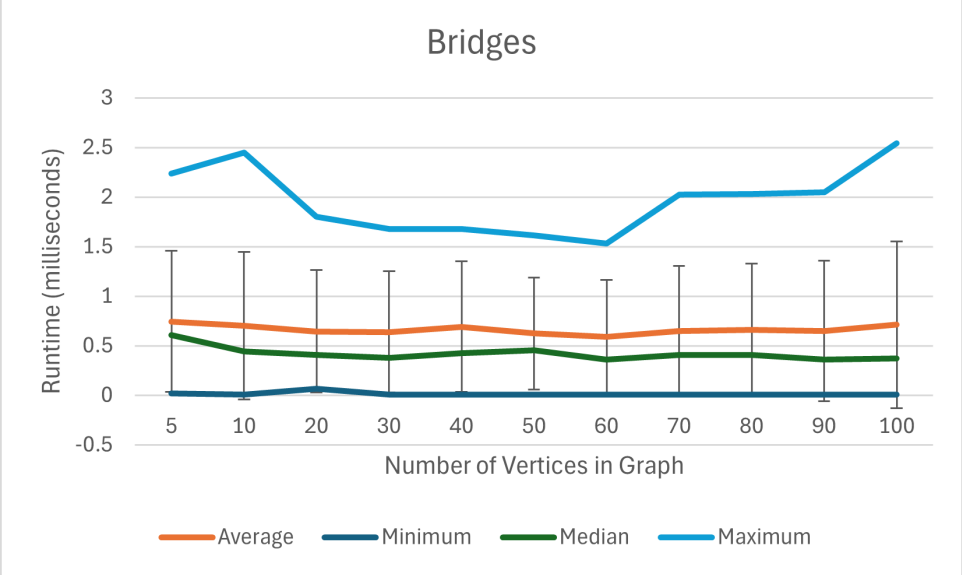

Figure 19: Runtime for the `AssertNBridges` constraint such that there are a number of edges between two `Subgraph`s equal to half the number of vertices in the `Graph`.

Inverse Floyd-Warshall was also run through *CatSAT* according to the following code written by Dr. Ian Horswill:

```
1 public void InverseFloydWarshall()
```

```
{
    Problem p = new Problem(name);
    Func<string, string> adjacent =
        Predicate<string, string>("
        adjacent");
    Func<string, string, int> floyd =
        Predicate<string, string, int
        >("d");
    Proposition D(string v1, string v2,
         int k) => k == 0 ? adjacent(v1
        , v2) : floyd(v1, v2, k);
    for (int k = 1; k < vertices.Length
        ; k++)
    {
        string vk = vertices[k];
        foreach (string v1 in vertices)
        foreach (string v2 in vertices)
            p.Assert(
                D(v1, v2, k) <= D(v1, v2
                    , k - 1),
                D(v1, v2, k) <= (D(v1,
                    vk, k - 1) & D(vk,
                    v2, k - 1))
            );
        }
        Proposition Connected(string v1
            , string v2) => D(v1, v2,
            vertices.Length - 1);
        foreach (string v1 in vertices)
        foreach (string v2 in vertices)
            if (v1 == v2 || (v1 !=
                vertices.Last() && v2 !=
                 vertices.Last()))
                p.Assert(Connected(v1,
                    v2));
            else
                p.Assert(Not(Connected(
                    v1, v2)));
        p.Optimize();
        Solution s = p.Solve();
        foreach (string v1 in vertices)
        foreach (string v2 in vertices)
            Assert.IsTrue(s[Connected(
                v1, v2)] == (v1 == v2)
                || (v1 != vertices.Last
                () && v2 != vertices.
                Last()));
    }
}
```

The table below demonstrates a comparison of line counts, and solution times (average, median, standard deviation, minimum, and maximum; all in milliseconds) for all of the above functions as well as Inverse Floyd-Warshall. Note that, for number of lines of code, we omit for loops that run through iterations of different numbers of vertices and number of solutions, as well as any comments. Also note that solutions were not found without *CatSAT* timing out for Inverse Floyd-Warshall for 20 vertices and greater.

Task		**SAT Problem**	**Solution Time (ms)**				
Description	**Code (lines)**	**Vertices**	**Average**	**STDev**	**Median**	**Min**	**Max**
`GraphConnected`	4	5	1.45	1.20	1.17	0.05	3.41
		10	1.17	1.01	0.95	0.04	3.16
		100	1.37	1.39	0.75	0.01	3.92
`InverseFloydWarshall`	27	5	17.79	8.80	17.79	8.99	26.59
		10	4.72	4.29	4.72	0.43	9.02
		100	-	-	-	-	-
`NodesConnected`	4	5	1.29	1.41	0.83	0.05	4.80
		10	1.12	1.37	0.68	0.03	4.77
		100	1.00	1.03	0.75	0.01	3.37
`TwoSubgraphsConnected`	9	5	1.91	1.49	1.43	0.10	4.39
		10	1.53	1.32	1.19	0.06	4.13
		100	1.13	0.99	0.82	0.01	3.31
`Cycle`	7	5	1.14	0.95	1.02	0.04	3.04
		10	1.02	0.90	0.95	0.01	2.85
		100	1.28	1.16	1.04	0.01	3.49
`Density`	4	5	0.96	0.95	0.57	0.02	2.90
		10	0.97	1.02	0.55	0.02	3.28
		100	0.82	0.91	0.51	0.01	3.08
`Bridges`	8	5	0.74	0.71	0.61	0.02	2.23
		10	0.70	0.74	0.44	0.01	2.45
		100	0.71	0.84	0.37	0.01	2.54

Table 1: Table number of lines of code, number of vertices, and solution time statistics for various graph constraints.

4 Applications

4.1 Potential Uses

Graphs are incredibly useful data structures in many areas of computer science and software development. The linking of similar kinds of data via specific relationships between this data is an abstraction that is particularly applicable in the fields of video game development and game AI. We present a non-exhaustive discussion of potential applications of graph constraint algorithms in video games.

One very direct application is the procedural generation of characters and the relationships between such characters. The nodes themselves may be characters and the edges the various relationships between characters. A designer could require that, for example, the character graph is connected, that a cycle exists within a subgraph (like a love triangle), or that two specific characters have not yet met but are friends of friends. Constraints like these could also be combined so long as they are not mutually exclusive. This application has been implemented with graph constraint algorithms in Section 4.2.

Another application similar to character graphs is entity/business graphs. Individual businesses would consist of densely connected subgraphs in an overall graph with a sparse number of bridges between clusters.

In both of the above examples, further work could be done to encode information and impose constraints along the edges in a graph. Type of relationship, strength of relationship, and direction of relationship could all become constraints and data used for both narrative and mechanic purposes in a video game. For more in-depth examples, see Section 4.2.

Graph constraint algorithms could also be used for map generation by use of waypoints. For example, density of the graph could be used as means to restrict or increase the different paths to go from city to city, islands can be represented with vertices of zero degree, and paths of certain length can be constructed to get from waypoint to waypoint.

The map waypoint generation method could also be applied to creating branching questlines for a story-based roleplaying games (RPGs). Note that some other tool would need to be used to generate content for the quests themselves, but graph constraint algorithms could be used to generate a graph representing the different paths the player of the game could take to complete the quests themselves.

4.2 *Why Can't We All Just Get Along?*

Why Can't We All Just Get Along? is a video game developed using Unity `2022.3.20f1 LTS` and C#, created with the intention of showcasing one possible application of graph constraint algorithms. It is largely inspired by *Reigns*, a strategy video game developed and published by Nerial, where the player is tasked with ruling a kingdom by accepting or rejecting propositions on cards while balancing statistics such as military power and money.

In *Why Can't We All Just Get Along?*, the player is thrown into a feud between two families and is required to make choices that ultimately lead to the families reuniting and re-establishing previously burnt bridges. Similar to the core gameplay loop

of *Reigns*, the player is presented with a card that depicts a particular scenario, such as, for example, a character inviting you to dinner. The player can then choose one of two options to advance the story, with each option potentially affecting both the player's statistics (reputation among their own family, money, and health/sanity) and the degree of compatibility between the two families.

In terms of technical architecture, *Why Can't We All Just Get Along?* relies a variety of procedurally generated content. The game's initial setup is developed such that each new instantiation and playthrough of the game presents a new cast of characters and family structures. First, *CatSAT* is used to generate the `Graph` corresponding to the two families (each being represented by a `Subgraph`) under the following imposed graph constraints:

1. Each family `Subgraph` must be connected (via a `SubsetConnectedConstraint`).

2. Each family `Subgraph`'s density must be between the specified minimum (`minDensity`) and maximum (`maxDensity`) values provided by the developer (via a `Density` constraint).

3. The `Graph` must be connected (via a `GraphConnectedConstraint`).

4. The number of bridge edges between the two family `Subgraph`s must be between the specified minimum (`minBridges`) and maximum (`maxBridges`) values provided by the developer (via a `AssertNBridges` constraint).

The relevant part of the code for this is shown below:

```
public CombinedFamily(int familyOneSize, int familyTwoSize, float minDensity, float maxDensity, int minBridges, int maxBridges)
{
    _problem = new Problem();
    _graph = new Graph(_problem, familyOneSize + familyTwoSize);
    _familyOneSubgraph = new Subgraph(_graph, Enumerable.Range(0, familyOneSize));
    _familyTwoSubgraph = new Subgraph(_graph, Enumerable.Range(familyOneSize, familyTwoSize));
    _familyOneSubgraph.AssertConnected();
    _familyTwoSubgraph.AssertConnected();
    _familyOneSubgraph.Density(minDensity, maxDensity);
    _familyTwoSubgraph.Density(minDensity, maxDensity);
    _graph.AssertNBridges(minBridges, maxBridges, _familyOneSubgraph, _familyTwoSubgraph);
    _graph.AssertConnected();
    _solution = _problem.Solve();
}
```

Once the graph has been generated and two surnames have been randomly chosen (one for each family), a number of characters (equal to the total number of nodes in the graph for the two families) must be randomly generated, using *Imaginarium*. The *Imaginarium* file used to specify the constraints for character generation is as follows:

```
Characters have an age between 18 and 70.

Characters are lawful good, neutral good, chaotic good, lawful neutral, true neutral, chaotic neutral, lawful evil, neutral evil, or chaotic evil.

Characters are any two of active, adventurous, affectionate, alert, ambitious, bold, bright, brave,
```

```
    calm, cheerful, clever, confident,
     cool, ...
6
7 Characters have an occupation from
    occupations.
8
9 Characters are feminine-named,
    masculine-named, or neutral-named.
10 Feminine-named characters have a first
     name from feminine first names.
11 Masculine-named characters have a
    first name from masculine first
    names.
12 Neutral-named characters have a first
    name from gender neutral first
    names.
13 Characters are identified as "[first
    name]".
14 Do not mention being feminine-named.
15 Do not mention being masculine-named.
16 Do not mention being neutral-named.
```

Here, note that `occupations`, `feminine first names`, `masculine first names`, and `gender neutral first names` are all text files provided to *Imaginarium* containing extensive lists of options. The characters are assigned to the nodes of the graph in ascending order of node number (an integer) and order of characters generated by *Imaginarium*.

The game setup then goes on to instantiate all possible scenario cards given the different card templates provided in `cards.json` and the characters generated by *Imaginarium*.

After all setup is complete, the player is shown an interactive menu where they can see the graphs generated by *CatSAT* and the aforementioned imposed graph constraints, as shown in Figure 20 in Section 4.2.1. The player is able to click on vertices in the graph to see more information (generated by *Imaginarium*) about the character corresponding to that vertex, which is displayed in the card on the right-hand side of the screen. The player is prompted to choose a character to play as, and the game begins to display cards with different scenarios presented as choices for the player to make, as their selected character (see Figure 21.

As *CatSAT*, graph constraints, and *Imaginarium* are all used to procedurally generate characters and family structures, each new playthrough of the game presents new play experiences. This particular application demonstrates the strengths of PCG to improve replayability as well as to significantly decrease the design work needed to create characters, families, and narratives in the game.

4.2.1 Screenshots

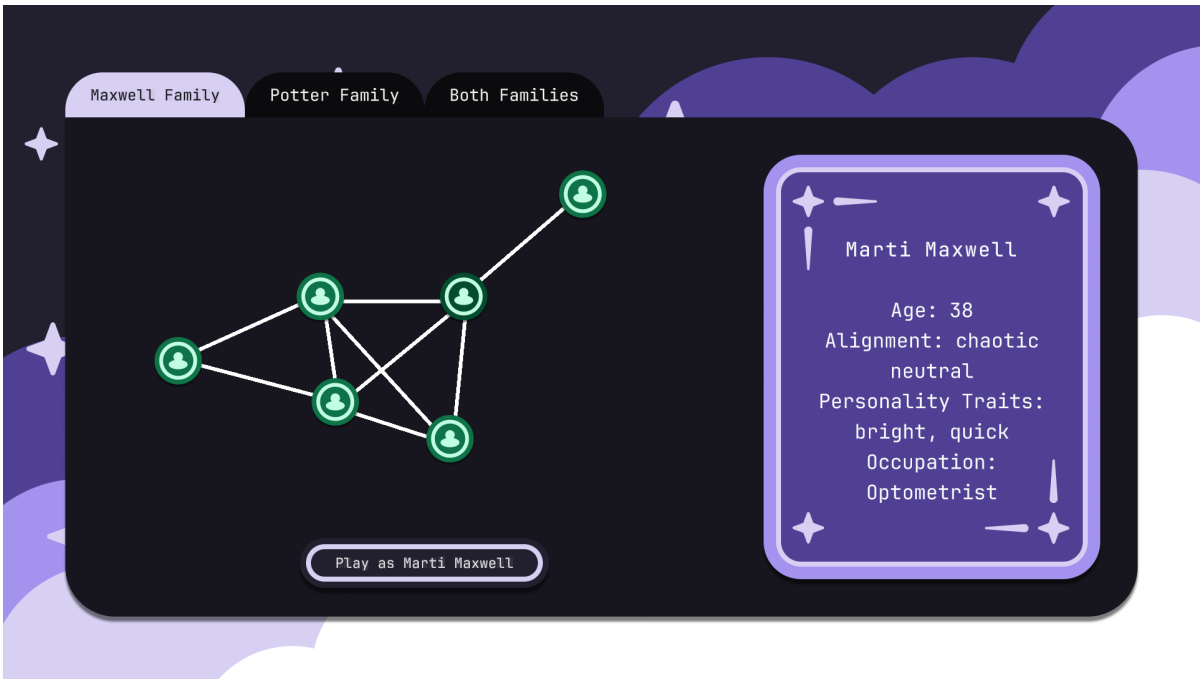

Figure 20: The setup screen in *Why Can't We All Just Get Along?*.

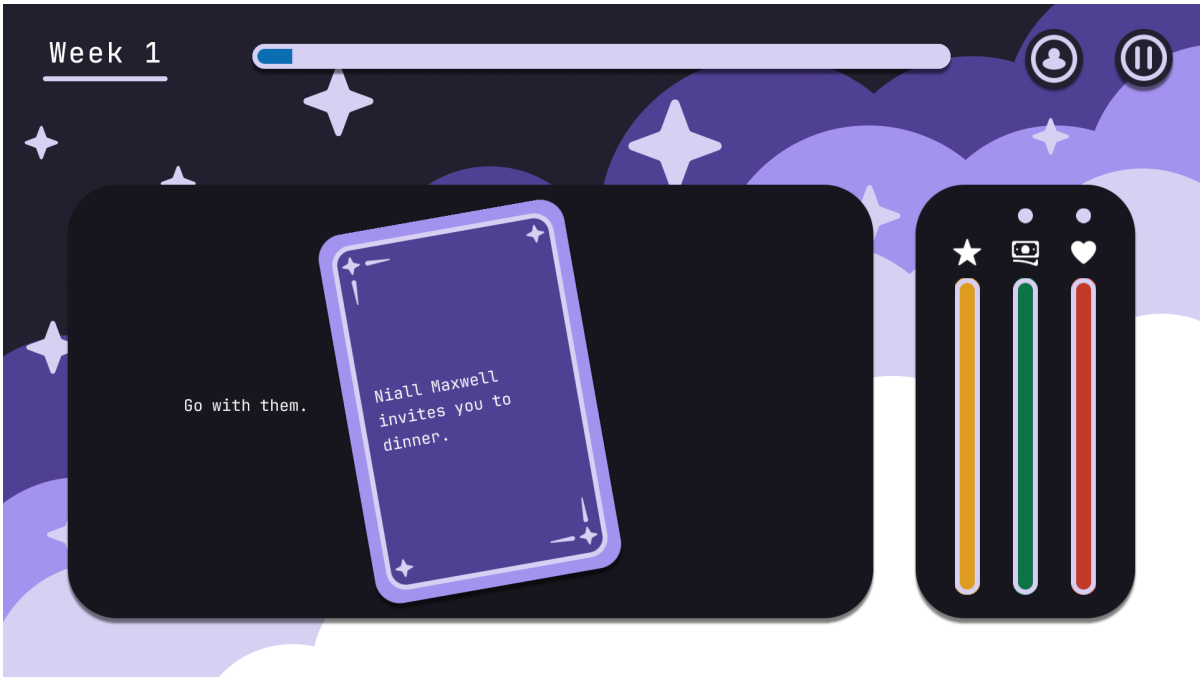

Figure 21: The main gameplay screen in *Why Can't We All Just Get Along?*.

5 Conclusion

SAT-based systems are outstanding to utilize in the field of PCG due to their flexibility, speed, and high level of expression available to the user. However, traditionally, it has been difficult to implement the generation of random connected graphs, let alone impose other kinds of constraints on a randomly generated graph. By expanding *CatSAT* with support for graph constraints such as connectivity, required paths between nodes, and density, users of the system can generate graphs as data structures for games in a timely manner by creating a problem and adding as many constraints as they desire. The expansion of *CatSAT* in this way also allows users to create constraints of their own, provided they have sufficient knowledge of logic programming, C# development, and graph theory. Problems for graphs can be solved in milliseconds with relatively few lines of code, with solutions that can be applied to a wide variety of content in video games.

5.1 Future Work

Further work can be done primarily in the following four ways:

1. Implementing new graph constraints.
2. Optimizing current graph constraints and improving upon the `GreedyFlip` algorithm.
3. Expanding the `Graph` and `Subgraph` implementations to incorporate features of graphs that do not exist in the current implementations.
4. Implement tools and GUIs for video game designers.

A presentation of ideas for each of the three aforementioned categories is given.

5.1.1 Implementing New Graph Constraints

As mentioned in Section 3.1, new constraints to be imposed on a `Graph` to be generated can be developed by creating a new C# class that is a subclass of `CustomConstraint`. The general class structure is given below:

```
1  public class NewGraphConstraint :
       CustomConstraint
2  {
3      public NewGraphConstraint(bool
           isDisjunction, ushort min,
           short[] disjuncts, int
           extraHash) : base (
           isDisjunction, min, disjuncts,
           extraHash) {}
4
5      public override int CustomFlipRisk(
           ushort index, bool newValue) {}
6
7      public override void
           UpdateCustomConstraint(
           BooleanSolver b, ushort pIndex,
            bool newValue) {}
8
9      public override bool IsSatisfied(
           ushort satisfiedDisjuncts) {}
10
11     internal override bool EquivalentTo
           (Constraint c) {}
12
13     internal override void Decompile(
           Problem p, StringBuilder b) {}
14
15     public override int
           ThreatCountDeltaIncreasing(
           ushort count) {}
16
17     public override int
           ThreatCountDeltaDecreasing(
           ushort count) {}
18
```

```
    public override void
        UpdateTruePositiveAndFalse
        Negative(BooleanSolver b) {}

    public override void
        UpdateTrueNegativeAndFalse
        Positive(BooleanSolver b) {}

    public override bool
        MaxFalseLiterals(int
        falseLiterals) {}

    public override bool
        MaxTrueLiterals(int
        trueLiterals) {}
}
```

Any class that inherits from `CustomConstraint` must override all of the methods written in the code snippet above.

The constructor for the new graph constraint must call the `base` constructor with parameters `bool isDisjunction`, `ushort min`, `short[] disjuncts`, and `int extraHash`. If applicable, the new constraint's constructor can accept additional parameters and initialize them accordingly in the body of the constructor.

`CustomFlipRisk` is a method that returns an integer associated with the risk or cost of flipping a specified SAT variable to a new value. The index of the SAT variable being flipped and the new boolean value of the literal is provided. For the integer that is returned, negative values correspond to a favorable flip, positive values correspond to an unfavorable flip, and zero corresponds to neither risk nor reward. This method is required for the `GreedyFlip` algorithm to pick a favorable variable to flip (in this case, to pick a favorable edge to add to or remove from the graph).

`UpdateCustomConstraint` is a method that updates the current solver's list of unsatisifed constraints when the value of a SAT variable is flipped. The method must be passed the current `BooleanSolver` (which contains the `UnsatisfiedClauses` list), the index of the SAT variable being flipped, and a boolean that is `true` if the edge corresponding to the SAT variable is being added to the graph, `false` if it is being removed from the graph. Any additional data relevant to the constraint should also be updated here. Implementation should look something like this:

```
public override void
    UpdateCustomConstraint(
    BooleanSolver b, ushort pIndex,
    bool adding)
{
    EdgeProposition edge = Graph.
        SATVariableToEdge[pIndex];
    if (adding)
    {
        // Update the graph accordingly
        // For example:
        Graph.ConnectInSpanningForest(
            edge.SourceVertex, edge.
            DestinationVertex);

        // Check if the constraint is
            now satisfied and
            previously was not
        if (constraintSatisfied && b.
            UnsatisfiedClauses.Contains
            (Index))
        {
            // Remove this constraint (
                which has index "Index")
                 from the list of
                unsatisfied constraints
            b.UnsatisfiedClauses.Remove
                (Index);
        }
    }
    else
    {
        // Update the graph accordingly
        // For example:
        Graph.Disconnect(edge.
            SourceVertex, edge.
            DestinationVertex);

        // Check if the constraint is
            now unsatisfied and
            previously was satisfied
```

```
24          if (constraintUnsatisfied &&
                previouslySatisfied)
25          {
26              // Add this constraint to
                    the list of unsatisfied
                    constraints
27              b.UnsatisfiedClauses.Add(
                    Index);
28          }
29      }
30  }
```

`IsSatisfied` is a method that is called once the problem being solved has been created with an initial random state, assessing whether the constraint has been satisfied without any variables being flipped. It is passed the number of satisfied disjuncts in the current solution.

`EquivalentTo` is a method that checks if the constraint is a copy of or is identical to the constraint passed to it.

`Decompile` is a function that creates a textual representation of the constraint, purely for debugging purposes. In most cases, it is useful to append the name of the constraint to the string builder:

```
1 internal override void Decompile(
      Problem p, StringBuilder b)
2 {
3     b.Append("NewGraphConstraint");
4 }
```

There are also a few functions which need to be overridden, however they are only relevant to pseudo-boolean constraints, so they will not be discussed here.

Below is a non-exhaustive list of constraints that could be implemented.

- `GraphCompleteConstraint`: The graph generated must be complete (every possible edge in the graph is present).
- `NConnectedComponentsConstraint`: The graph generated must have a specified number of connected components, n.
- `StronglyConnectedComponent Constraint`: The graph generated must contain a strongly connected component.
- `NodesNotConnectedConstraint`: The vertices n and m in a graph must not be connected via any path.
- `PathOfLengthConstraint`: If possible, the path between the vertices n and m in the graph must be of a specified length, ℓ.
- `CycleConstraint`: The graph generated must have a cycle.
- `CycleOfLengthConstraint`: The graph generated must have a cycle of a specified length, ℓ.
- `PathThroughVertexConstraint`: The vertices n and m must be connected via a path that passes through a third specified vertex, v.
- `TreeConstraint`: The generated graph must be acyclic and connected.
- `ForestConstraint`: The generated graph must only consist of connected components that are acyclic.
- `GraphColoringConstraint`: The graph generated must be properly colored with a specified number c of colors.
- `IndependentSetConstraint`: The graph generated must contain a maximum independent set, where no two vertices in the set are adjacent.

- `GraphSymmetryConstraint`: The graph generated must have specified symmetry (mirror, rotational, etc.).
- `PlanarGraphConstraint`: The graph generated must be planar (edges can be drawn such that no pair of edges intersects except for at endpoints).

5.1.2 Optimizing Current Graph Constraints

Many of the current graph constraints can be optimized by means of caching relevant information, making `CustomFlipRisk` assessments more greedy, and improving the `GreedyFlip` algorithm to choose even more favorable SAT variables to flip. `RebuildSpanningForest` in the `Graph` and `Subgraph` classes can be improved by maintaining a list or set of edges that were present in the spanning forest before clearing it to iterate over instead of compiling this list from the current solver. The `GreedyFlip` method in `GraphConnectedConstraint` can be modified to discourage removing edges that are present in the graph's current `SpanningForest`. Further advancements can be made by using more efficient and elegant search algorithms, at the developer's discretion.

5.1.3 Expanding `Graph` and `Subgraph`

CatSAT's graph constraints currently only support graphs with unweighted and undirected edges, at the time of writing this paper.

For directed edges, consider maintaining degree information for every vertex in the graph, particularly keeping track of in-degree, out-degree, and overall degree. For instance, vertex degrees could prove useful in `NodesConnectedConstraint` (see Section 3.3.3) as the source vertex should have in-degree `0` and out-degree `1`, the destination vertex should have in-degree `1` and out-degree `0`, and every other vertex in the graph should have nonzero in- and out-degrees (else it is not in the path).

Note that the `EdgeProposition` class (see Section 3.2.1) would likely need to be modified with a field indicating whether the edge is undirected or directed. For edge weights, consider adding a `float` or `double` field to the `EdgeProposition` class. In either case, any pre-existing or newly added search algorithms would need to be modified to account for edge direction and/or weight. In current implementations of graph constraints, solely the breadth-first search in the `ShortestPath` method of the `NodesConnectedConstraint` (see Section 3.3.3) would need to be modified.

5.1.4 Tools and GUI

Using *CatSAT* in a Unity project requires at the very least a basic understanding of programming in C#, as *CatSAT* is imported into Unity as a DLL file, which gives the developer access to all public methods and variables. As a result, the user needs to write code according to the instructions provided in Section 3.1 in order to create a problem and impose graph constraints on said problem. Although this process should be simple and straightforward for game programmers, it may prove less accessible for someone without any coding experience or knowledge. It would be very ideal for someone like a game designer to have a Unity `EditorWindow` that has an intuitive graphical interface letting them create graphs, add constraints, and modify variables when appropriate.

5.2 Links to Repositories and Work

The *CatSAT* repository can be found here, with the branch containing graph constraints here. The *ImaginariumCore* repository can be found here, and the work-in-progress *Why Can't We All Get Along?* repository can be found here. An interactive website will also be under development, to be completed by the end of the Spring 2024 Quarter, found here.

5.3 Acknowledgements

I would first and foremost like to thank Dr. Ian Horswill for the utmost care, patience, kindness, and mentorship he has given me over the past few years. I fondly remember meeting him as my professor in my first ever computer science course at Northwestern and immediately knowing that he was someone I wanted to work with and grow with as a student, game developer, and person. In addition to always sharing his seemingly infinite wealth of knowledge about any subject one could imagine, Dr. Horswill has consistently been eager and willing to have thought-provoking conversations about similar passions, provide academic and career-related advice, and offer emotional support in times of stress. For all of this, I am extremely grateful; I would not be the student, game developer, and person that I currently am without him.

I would also like to thank my committee members, Dr. Robert Zubek and Dr. Simone Campanoni, for their patience and willingness to help and support me with my work for this thesis. This process has been one of trials and tribulations, and their advice and enthusiasm have made the world of a difference and bolstered my perseverance as I have completed my work.

I would like to thank the incredibly talented students in Dr. Horswill's research group: Eleftheria Beres, Jack Burkhardt, Aryaman Chawla, Steven Gu, Samuel Hill, Jason Li, Chenyu Wang, and Sorie Yillah. I am truly blessed to have worked with and grown close to them over the past two years. I will forever treasure our weekly game nights, the invaluable help and advice they have given me, and their willingness to listen to me go on and on about my passions and work. I have never known a more compassionate, hilarious, and intelligent group of people and I cannot wait to see the work that they all complete in their academic careers and beyond.

I would also like to thank my friends, who have been there for me over the past few years, have joined me for countless long work nights, and have given me unconditional support and love. Regardless of whatever may have been going on in my life, I have always been able to say that I have the best friends in the entire universe and the most strong support system I could ever ask for, and for that I owe them everything. I would never have been able to accomplish all that I have without their relentless encouragement, caring words, and hours upon hours of time spent together.

As an aside, I would like to thank Toby Fox, creator of the game *Undertale*, for composing the soundtrack that fueled the writing of this thesis. At the time of writing, I have listened to "CORE" over 150 times. Thank you to Toby Fox for making a spectacular game and an absolutely phenomenal soundtrack.

Finally, I would like to thank my mother and father. They have done everything for me, from supporting my education from day one to encouraging me to explore my

passions, and I am forever indebted to them. I will forever and always appreciate the love, advice, and patience they have given me unconditionally throughout my entire life, but especially throughout my academic career as I have grown into the person I am.

References

[1] G. Smith, E. Gan, A. Othenin-Girard, and J. Whitehead, "Pcg-based game design: Enabling new play experiences through procedural content generation," 06 2011.

[2] R. Hunicke, M. Leblanc, and R. Zubek, "Mda: A formal approach to game design and game research," *AAAI Workshop - Technical Report*, vol. 1, 01 2004.

[3] I. Horswill, "Answer set programming for pcg: the good, the bad, and the ugly," vol. 3217, 2021, Conference paper, cited by: 1. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85139091221&partnerID=40&md5=17c209196582007c325593d489b3d441

[4] A. Summerville, C. Martens, B. Samuel, J. Osborn, N. Wardrip-Fruin, and M. Mateas, "Gemini: bidirectional generation and analysis of games via asp," in *Proceedings of the Fourteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, ser. AIIDE'18. AAAI Press, 2018.

[5] O. Polozov, E. rourke, A. Smith, L. Zettlemoyer, S. Gulwani, and Z. Popovi´c, "Personalized mathematical word problem generation," 07 2015.

[6] B. Selman, H. Kautz, and B. Cohen, "Local search strategies for satisfiability testing," *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, vol. 26, 09 1999.

[7] H. Hoos, "An adaptive noise mechanism for walksat," *Proceedings of the National Conference on Artificial Intelligence*, 08 2004.

[8] N.-F. Zhou, "Modeling and solving graph synthesis problems using sat-encoded reachability constraints in picat," *Electronic Proceedings in Theoretical Computer Science*, vol. 345, p. 165–178, Sep. 2021. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.345.30

[9] M. Bodirsky, C. Gröpl, and M. Kang, "Generating labeled planar graphs uniformly at random," *Theoretical Computer Science*, vol. 379, no. 3, pp. 377–386, 2007, automata, Languages and Programming. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397507001491

[10] W. McColl and K. Noshita, "On the number of edges in the transitive closure of a graph," *Discrete Applied Mathematics*, vol. 15, no. 1, pp. 67–73, 1986. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0166218X8690020X

[11] J. W. D. Jr., "The compactness of first-order logic:from gödel to lindström," *History and Philosophy of Logic*, vol. 14, no. 1, pp. 15–37, 1993. [Online]. Available: https://doi.org/10.1080/01445349308837208

[12] I. Horswill, "Catsat: A practical, embedded, sat language for runtime pcg," *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 14, pp. 38–44, 09 2018.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. Mit Press, 1990.