Map Sketch Generation as a Service

Antonios Liapis
Institute of Digital Games
University of Malta
antonios.liapis@um.edu.mt

Abstract

This paper describes the structure of a webservice able to generate simple game levels via constrained evolutionary optimization. The provided webservice allows users to generate playable game levels without needing to understand the underlying process and without having to allocate computational resources for doing so; combined with the highly expressive and customizable generator, a broad range of levels for different genres and purposes can meet many user needs.

Introduction

Research in Procedural Content Generation (PCG) for games often focuses on the design and implementation of generators for a specific game such as *Starcraft* (Togelius et al. 2010) and *Civilization* (Barros and Togelius 2015), or for a genre such as action-adventure games (Dormans and Bakkes 2011) or arcade games (Lim and Harrell 2014). Such generators are usually domain-specific and closed-source; their findings are elaborated in the accompanying papers, while the generators themselves are not readily available to interested readers. Among the rare instances where level generators have been open-sourced and re-used by the research community, the generator for *Infinite Mario Bros* is worthy of note as it has been extensively used as a basis of further research and competitions (Togelius et al. 2013) — admittedly in part due to the original game's popularity.

Making a generator publicly available as open-source software is often a lower priority due to, in part, the need for the original developers to provide sufficient documentation of the code so that interested users can understand its inner workings. However, research (and its accompanying code) rarely follows a straight path from questions to answers; even more critically, the code for research projects is not built on or adheres to a priori defined software specifications. The generator often acts as a demonstrator or case study of the ideas in accompanying papers; it is thus understandable that code clarity, interoperability, and usability (in the case of human-computer interfaces) take a back seat to the novelty of the underlying algorithmic technique. Even if research projects are released as open-source software, however, their adoption by end-users or other researchers is hindered by requirements for obscure code libraries or for specific operating systems and development environments. In addition, interested users often need to thoroughly read the available code and its documentation, which is made more difficult as it integrates novel, cutting-edge research.

A potential solution to the problem of making research output available to a wider audience without the need for the audience itself to fully understand it (and for the developers to fully document it) is the *webservice*. According to the World Wide Web Consortium (2004), "web services provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks". Among the most important properties of webservices is that (a) they are self-contained and self-describing, and (b) they communicate via open protocols; the former allows a webservice to have its own underlying architecture (including complex algorithms in obscure languages, as often found in research prototypes) and the latter allows data to be passed to and from a webservice in a straightforward, standardized form which allows a webservice to link with other software or other webservices.

This paper describes how an evolutionary system capable of generating small-scale game levels, identified as *map sketches*, is integrated into a webservice. The low-resolution abstract map sketches can represent a broad range of game levels for different genres by customizing the tiles included in each map sketch. This makes the generator highly expressive and thus appealing to a broad audience and use cases; therefore it is appropriate for release as a publicly available webservice. The webservice method calls allow end-users or other software to request the generation of new game levels, to specify game levels as seeds for further optimization, or to evaluate game levels passed along as arguments. The webservice comes with many fitness functions and constraint tests, which guide the generative process or can be used to assess the quality of generated or authored game levels.

Generating Map Sketches for Games

A map sketch is a low-resolution, high-level abstraction of a game level. Map sketches are sufficiently simple both for a human designer and for a genetic algorithm to optimize. Map sketches are typically comprised of a small number

of tiles, arranged on a grid; tiles can represent fundamental gameplay properties for specific game genres such as bases for strategy game levels, monsters for rogue-like dungeons, weapon pickups for first person shooters etc. Map sketches have been used in Sentient Sketchbook (Liapis, Yannakakis, and Togelius 2013a) as the target of a human-machine co-creative process (Yannakakis, Liapis, and Alexopoulos 2014), where map sketches are generated by the computer as alternatives to human designs. Several computationally lightweight fitness functions have been defined by Liapis, Yannakakis, and Togelius (2013b) for evaluating human designs and as objectives for genetic algorithms. Map sketches represent game levels with playability constraints; these constraints are satisfied through a constrained evolutionary optimization algorithm described below.

Evolutionary algorithm details

The evolutionary algorithm used to generate map sketches performs constrained optimization: the map sketches need to fulfill some minimal criteria in order to be playable, while playable map sketches should possess certain desirable features. Evolution is performed via a feasible-infeasible two-population genetic algorithm (FI-2pop GA), which evolves infeasible individuals (which fail any constraints) in a separate population to feasible ones (Kimbrough et al. 2008). Feasible individuals evolve to optimize a domain-specific fitness function, while infeasible individuals evolve to minimize their distance from feasibility; infeasible individuals close to the feasible border are more likely to create feasible individuals (Schoenauer and Michalewicz 1996). Feasible offspring of infeasible parents migrate to the feasible population (and vice-versa), thus allowing a form of interbreeding which is likely to increase the diversity of both populations. Evolution is performed via asexual mutation alone, or via crossover and mutation (De Jong 2006). If mutation is applied, the mutated offspring transforms several of its tiles; mutation can swap adjacent tiles or change a tile's type to another type (e.g. change walls to empty tiles and vice versa). All parameters of the evolutionary algorithm can be defined in the input of the webservice method calls.

Evaluating sketches

Map sketches must fulfill certain criteria in order to be minimally playable, and must also have certain qualities in order to be useful in a game. In earlier work (Liapis, Yannakakis, and Togelius 2013b), the core playability criterion for map sketches was that all level features can be reached, i.e. there should be a path between all tiles with gameplay properties ("special" tiles). Other criteria include limits to the number of certain tiles (e.g. a dungeon can only have one entrance and one exit). For feasible map sketches, the general level design patterns (Björk and Holopainen 2004) of *symmetry*, *exploration* and *area control* were used as inspiration for fitness functions which evaluate resource control, area control and exploration as well as their balance dimensions (e.g. the balance of resource control of two player bases in a strategy game map). Resource control and area control are derived from each tile's *safety score*: a tile's safety score to one special tile is high if it is much closer to that special tile than to any other special tile of this type. On the other hand, exploration is evaluated based on a flood fill algorithm starting from one special tile and stopping when another special tile has been reached; the *exploration score* averages the exploration effort from one special tile to all other special tiles of this type. More information on the map sketch evaluations can be found in previous publications (Liapis, Yannakakis, and Togelius 2013b).

Webservice structure

The webservice can generate new map sketches, create variations of provided map sketches, or evaluate existing map sketches; all of these functionalities will be described later in this section. The webservice communicates with other applications (clients) using data structures in the JSON (JavaScript Object Notation) format, which is a lightweight data-interchange format easily understandable by both humans and machines. For all method calls, a client application must provide as input (via a POST request) a JSON object containing the specifications of the map sketch, the desired qualities and constraints on the generated or evaluated sketches, and (optionally) any parameters needed for the genetic algorithm; the webservice processes this input and returns a JSON data structure as a result, the format of which depends on the method call made. The input JSON object can customize the generation or evaluation properties of the method call; the next sections describe its components.

Defining a map sketch

A map sketch consists of tiles of different types laid on a grid. The input JSON object contains a `TileTypes` value which is an array of tile type definitions; each definition includes the tile type's `name` which acts as its unique identifier (for e.g. fitness evaluations), its `asciiChar` for producing the ASCII map which is output by the webservice, a `passable` flag which is false if the tile type blocks movement and an optional `defaultTile` flag (presumed false if missing). Only one tile type can be the `defaultTile`; a newly initialized map consists primarily of this type of tiles.

Defining fitness functions and constraints

In order to evaluate map sketches (e.g. when optimizing them through evolution, or when testing if their playability constraints are met), multiple fitnesses and constraints can be defined in the input JSON object in a `Fitness` array and a `Constraints` array respectively. Each fitness or constraint definition has a `name` for displaying the map sketch's fitness scores in evaluation method calls, a `type` which specifies the algorithms used to evaluate the fitness score, an optional `weight` which specifies the impact of this dimension to the overall fitness score (presumed 1 if missing), `referenceTiles` and `targetTiles` which specify which tile types are considered or targeted (respectively) when evaluating this fitness score and `arguments` which includes any additional arguments for this type of fitness.

Among the parameters of a fitness or constraint definition, the `type` is most important as it specifies the algorithms

used to calculate the fitness score but also how all other parameters (except `name` and `weight`) will be interpreted. For instance, `targetTiles` or `arguments` are necessary for certain types of fitnesses and constraints, but can be omitted in others. Each of the `referenceTiles`, `targetTiles` and `arguments` is a text string which may contain multiple parameters separated by commas (the text string is split within the webservice). The different fitness functions and constraints are described below, according to their `type`:

`SafeAreaThresholdFitness` evaluates the number of tiles which are safe to any `referenceTiles`. If the fitness definition has a `targetTiles` parameter, then only tiles of the types in `targetTiles` are considered; otherwise all passable tiles are considered. The `arguments` parameter defines the lowest safety score for a tile to be considered safe.

`SafeAreaThresholdBalance` uses the same structure as `SafeAreaThresholdFitness` and evaluates if each of the `referenceTiles` has a similar number of safe passable tiles (if `targetTiles` is omitted) or safe tiles of a type found in the `targetTiles` parameter.

`TileSafetyFitness` evaluates the total safety score of all `targetTiles` with regards to all `referenceTiles`.

`TileSafetyBalance` evaluates if the cumulative safety score of all `targetTiles` for each of the `referenceTiles` is similar.

`ExplorationFitness` evaluates the exploration effort to discover all `targetTiles` from all `referenceTiles`. If `targetTiles` are omitted, the fitness evaluates the exploration effort from all `referenceTiles` to all other `referenceTiles`.

`ExplorationBalance` evaluates if the exploration effort is similar for each of the `referenceTiles` when discovering all of the `targetTiles` or when discovering all other `referenceTiles` (if `targetTiles` are omitted).

`ConnectivityConstraint` measures how many of the `targetTiles` are not connected via a passable path to `referenceTiles`; if `targetTiles` are omitted, this constraint enumerates the disconnected paths between all `referenceTiles`. Optionally, if the `arguments` has a `"disconnected"` value then the constraint is only satisfied if no passable paths exist between the specified tiles.

`NumericalConstraint` measures the number of excess or missing `referenceTiles` as specified in the `arguments`. The `arguments` parameter can specify a number (`A`) or two numbers (`A,B`) as the numerical bounds: the possible syntax for `arguments` is `"equals,A"`, `"notEquals,A"`, `"maximum,A"`, `"minimum,A"`, `"inRange,A,B"`, `"notInRange,A,B"`.

`ConditionalConnectivityConstraint` is similar to `ConnectivityConstraint` but enumerates disconnected paths as if certain tile types were impassable or passable (overriding specifications in `TileTypes`). Passability/impassability is specified in the `arguments`, where a tile type is preceded by "passable" or "impassable" (e.g. for a map sketch with "wall" and "treasure" tile types, this constraint could have an `arguments` value of `"passablewall,impassabletreasure"`).

`DistanceConstraint` enumerates those shortest paths between `referenceTiles` and `targetTiles` which are within numerical bounds specified in the `arguments` as per `NumericalConstraint`. If `targetTiles` is omitted, paths between all `referenceTiles` are considered.

Defining evolutionary parameters

The input JSON object can include evolutionary parameters as a `Parameters` object, used for generating new map sketches. Some parameters are necessary for any generation method call, while optional parameters can override the system's default values used for evolution. Necessary parameters are `runs` (i.e. the number of evolutionary runs and the number of returned map sketches), `mapSizeX` and `mapSizeY` which specify the output maps' dimensions, `population` (i.e. the number of evolving feasible or infeasible individuals) and `maxGenerations` of evolution before the webservice returns its output map sketches.

Optional parameters override default values used by the webservice, and include the `fi2pop` flag which, if false, uses a single population which applies the death penalty (i.e. a fitness of 0) to infeasible individuals for evolution (default is true), `steadyPercentage` (i.e. the number of best individuals copied to the next generation), `crossoverPoints` (i.e. the N in the N-point crossover used by GAs), `mutateOnlyProbability` (i.e. the chance in 100 of asexual mutation), `mutateAnyProbability` (i.e. the chance in 100 of mutating an offspring of two parents). For mutation, `mutateTileMinNumber` and `mutateTileMaxNumber` describe the range in the number of tiles changing in each mutation cycle, `mutateShift` specifies the chance in 100 of any mutation swapping adjacent tiles and `mutateToggleA` specifies the chance in 100 of any mutation changing a tile of type A (A being the tile type's `name`) to the `defaultTile` and vice versa. At the system's default values for mutation, there is no chance of any tile type changing to another tile type but there is a 5% chance of swapping adjacent tiles.

Evolving a map sketch

Map sketches can be evolved via the `sketchgenerator` method call; this call returns the fittest feasible individual of an evolutionary run, or the fittest individuals of multiple runs (one individual per run). The client provides a JSON object as input (via a POST request) which must contain values for `TileTypes`, `Fitness`, `Constraints` (even if it is empty) and `Parameters` with the essential parameters described above. Optionally, the input JSON object can also contain a `ReferenceTileMaps` value as an array of one or more sketches which seed evolution; the sketches are provided as ASCII text, with characters defined in the accompanying `TileTypes`; rows are divided by semi-colons. Note that the map size of `ReferenceTileMaps` overrides the `mapSizeX` and `mapSizeY` parameters (which can be omitted): evolved sketches will have the same map size as the `ReferenceTileMaps`. The output of the evolutionary process is a JSON array containing one or more sketches

in ASCII text in the same format as ReferenceTileMaps. The number of sketches should be equal to the runs value of the Parameters object; however, some runs may not result in feasible individuals — e.g. in cases of highly constrained search spaces, few generations of evolution, or poorly designed constraints. In such cases, the number of sketches in the output may be fewer than the number of runs, or the array may even be empty.

Evaluating a map sketch

Evolution requires that every map sketch is tested for constraint satisfaction and also evaluated on the provided fitness scores; however, the results of evolution method calls do not contain such information for the sake of readability and bandwidth. Generated or custom-made sketches can be evaluated via the same fitness functions and constraints used internally in the generator. Two method calls allow for such an evaluation: sketchevaluator and sketchdetailevaluator, the latter offering all the functionalities of the former but with additional feedback for visualizing properties of the map sketch. For both methods, the client provides a JSON object as input (via a POST request) with a ReferenceTileMaps array of map sketches to be evaluated (in the same format as the evolution method call), along with TileTypes, Fitness and Constraints. The output of sketchevaluator is a JSON array of the same size as the input ReferenceTileMaps. Each item of the array is a JSON object which includes a feasible boolean value which is true if the map sketch satisfies all constraints and a scores array containing the map's scores for all fitnesses in the Fitness collection, if the map is feasible, or the map's distance from feasibility for all constraints in the Constraints collection, if the map is infeasible. Within the scores collection, each score is preceded by the name of the fitness or constraint being evaluated. In addition to the feasible and scores arguments, the output contains a parsedInput argument; for sketchevaluator method calls, parsedInput only contains the asciiMap argument with the same map sketch as provided in the input JSON object (for easier reference or asynchronous requests). For sketchdetailevaluator method calls, the parsedInput argument of each map contains (beyond asciiMap) a layers object containing several maps used internally by the generator; these maps can contain boolean (0 or 1) or floating point values (with each number separated by commas); map rows are divided by semi-colons. The names and types of maps depend on the constraints and fitnesses used for evaluation.

Examples

To better demonstrate how the webservice inputs and outputs data, several method calls for generating and evaluating map sketches of different types are included in Tables 1–3 along with the responses of those calls. Customizable examples of these method calls can be found online at: www.sentientsketchbook.com/webservice.php.

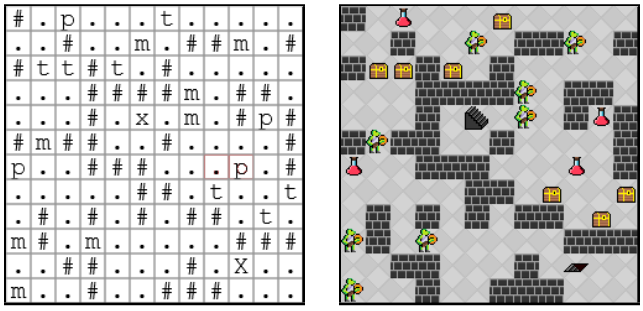
Table 1 showcases a method call which generates levels for the *MiniDungeons* game. *MiniDungeons* is a simple puzzle rogue-like game developed primarily for the purposes of modeling human decision-making (Holmgård et al. 2014). Levels of *MiniDungeons* contain empty passable tiles, impassable walls, potions, monsters, treasure, an entrance and an exit: these are defined in the TileTypes array, with the defaultTile being empty (i.e. all levels start with mostly empty tiles). Constraints for playable levels are found in the Constraints array, and include numerical constraints (of the type NumericalConstraint) on exits (only one exit allowed, as per the "equals,1" value of arguments), entrances, monsters, treasures and potions. A constraint that all entrance, exit, monster, potion and treasure tiles are connected via passable paths is added as connectAll, where the referenceTiles parameter has a comma separated list of those tile types' names. The fitness dimensions for optimizing *MiniDungeons* levels are found in the Fitness array: a TileSafetyFitness evaluates how safe treasure tiles (in targetTiles) are to monster tiles (in referenceTiles) and a TileSafetyBalance evaluates how balanced treasure safety is among monsters; a SafeAreaThresholdFitness evaluates how much area is controlled by each monster tile, entrance tile and exit tile (in referenceTiles) considering only tiles above a safety score of 0.35 (in arguments), and a SafeAreaThresholdBalance evaluates how balanced the safe areas are among these tiles; an ExplorationFitness evaluates how easy it is to discover the exit (in targetTiles) starting from the entrance (in referenceTiles), using only the cardinal directions (noDiagonals value in arguments). Finally, the generative parameters (i.e. Parameters) specify 5 optimization runs (thus resulting in 5 feasible *MiniDungeons* levels as the best of each run), a level size of 12 by 12 tiles, and results returned after 50 generations of evolution using a total of 50 feasible and infeasible individuals. Evolution is carried out only via asexual mutation (as mutateOnly is 100 out of 100 and thus certain); beyond standard mutation operators, an additional mutation which converts wall tiles to empty tiles (the default tile type) and vice versa is specified (with a 5% chance of occurring as per mutateTogglewall). The response of the webservice (at the bottom of Table 1) is a JSON array of 5 ASCII strings representing the 5 best feasible levels from the 5 optimization runs. The resulting ASCII strings use the same notation for tiles as in the asciiChar parameter of each tile type in the TileTypes; see Fig. 1 for a visualization of the first ASCII string in the array.

Table 2 showcases a method call which generates map sketches for shooter games, with a custom map sketch used as the seed for evolution. These map sketches contain empty passable tiles, impassable walls, team spawn points, weapons and healthpacks, as defined in the TileTypes array. Similar to *MiniDungeons* levels, the Constraints array specifies numerical constraints on spawnpoints, weapons and healthpacks as well as a ConnectivityConstraint that those tile types are all connected via passable paths. The Fitness array includes an ExplorationFitness which evaluates the discovery effort of spawnpoints and weapons starting from any other spawnpoint or weapon (in referenceTiles) and an ExplorationBalance

```
{"TileTypes":[{"name":"empty","asciiChar":".","passable":"true","defaultTile":"true"},{"name":"wall","asciiChar":
"#","passable":"false"},{"name":"entrance","asciiChar":"x","passable":"true"},{"name":"exit","asciiChar":"X","p
assable":"true"},{"name":"potion","asciiChar":"p","passable":"true"},{"name":"treasure","asciiChar":"t","passa
ble":"true"},{"name":"monster","asciiChar":"m","passable":"true"}],"Constraints":[{"name":"connectAll","type":
"ConnectivityConstraint","referenceTiles":"monster,potion,treasure,entrance,exit"},{"name":"monsterNumber","type":
"NumericalConstraint","referenceTiles":"monster","arguments":"equals,8"},{"name":"potionNumber","type":"Numerical
Constraint","referenceTiles":"potion","arguments":"equals,4"},{"name":"treasureNumber","type":"NumericalConstraint
","referenceTiles":"treasure","arguments":"equals,7"},{"name":"entranceNumber","type":"NumericalConstraint","refer
enceTiles":"entrance","arguments":"equals,1"},{"name":"exitNumber","type":"NumericalConstraint","referenceTiles":
"exit","arguments":"equals,1"}],"Fitness":[{"name":"treasureSafety","type":"TileSafetyFitness","referenceTiles":
"monster","targetTiles":"treasure"},{"name":"treasureSafetyBalance","type":"TileSafetyBalance","referenceTiles":
"monster","targetTiles":"treasure"},{"name":"exit+monsterArea","type":"SafeAreaThresholdFitness","referenceTiles":
"exit,entrance,monster","arguments":"0.35"},{"name":"exit+monsterAreaBalance","type":"SafeAreaThresholdBalance","
referenceTiles":"exit,entrance,monster","arguments":"0.35"},{"name":"exitExploration","type":"ExplorationFitness"
,"referenceTiles":"entrance","targetTiles":"exit","arguments":"noDiagonals"}],"Parameters":{"runs":5,"mapSizeX":
12,"mapSizeY":12,"maxGenerations":50,"population":20,"mutateTogglewall":"5","mutateOnly":"100"}}
```

```
["#.p...t.....;..#..m.##m.#;#tt#t.#.....;...####m.##.;...#.x.m.#p#;#m##..#....#;p..###...p.#;.....##.t..t;.#.#.#.##.t
.;m#.m.....###;..##...#.X..;m..#..###...","##...#.#...m;..m###..###.;##t.tx...m..;..#......##.;..##.p###.m.;.m...#.#
###.;p.##..#p.###;.##.#.#.##..;.#m....p#.m.;tm#.#.t.....;t##..###.###;.#.t.#.t...X","..#.##m##...;.###.#....#.;#Xp..
.#x.##.;###t..#.#.t.m;.#t.m.###.#.;##.m..#.#.#p;###p#.t##p##;.##mtm.t...#;#.##m##.#...;######.#.##.;##.##....t..;#..#
#....m#.","X.#.#....mt#;#..m#...####;.#..##.#....#;#.p.##...#.m;.p.#t.#m.#..;#..##.t.##.#;..#.#.###..#;#t..#.#p#tmt;#
m#.#.#.###.;#..#.##.##..;##..m#.m..x#;..#....#.tp#","##...#.####.;##.m.m#.###.;##.....m#.#;.x.##.##..#p;p.t.#..###.
.;.#..#.#t.ppt;#tm##..m#...;.#..#.#.##..;mm.#.#..##.#;#t..#..##t##;..##........;##.m.t#...X."]
```


Table 1: Generation JSON for *MiniDungeons* levels, with the webservice response below it (5 levels, best of 5 runs).



(a) ASCII map (b) Level for *MiniDungeons*

Figure 1: A 2D visualization of the first ASCII map from the result JSON array of Table 1.

evaluating whether that discovery effort is balanced; a `SafeAreaThresholdFitness` evaluates the extent of passable tiles which are considered safe by healthpacks (of any positive safety score due to 0.0 in `arguments`), and `SafeAreaThresholdBalance` evaluates if it is balanced among all healthpacks. The map seed, copies and permutations of which constitute the initial population for evolution, is found in the `ReferenceTileMaps` array as an ASCII string. The generative parameters (i.e. `Parameters`) specify one optimization run; no map size information (`mapSizeX` and `mapSizeY`) is provided, as those parameters are collected from the `ReferenceTileMaps`. Optimization uses a single population with infeasible individuals assigned a fitness of 0 (since `fi2pop` is false), and evolves for 100 generations on a population of 50 individuals; evolution is carried out via 2-point crossover and 5% chance of mutating the offspring, as per the default evolutionary parameters, while the `mutateTogglewall` parameter adds a chance of mutating wall tiles to the default empty tiles and vice versa. The response of the webservice (at the bottom of Table 2) is an array of one ASCII string (best of one optimization run); the resulting map sketch shares some structural similarities with the original map sketch used as a seed (see e.g. the spawnpoint positions which are identical), but 100 generations of optimization have caused changes via tile shifting or wall addition due to the custom mutation operator.

Table 3 showcases a method call which evaluates two map sketches for 4X strategy games[1] similar to *Civilization IV* (Firaxis 2005). These map sketches contain land tiles, water tiles and mountain tiles (the latter two being impassable) as well as city tiles (denoting players' starting cities), horse resource tiles and iron resource tiles; these tile types are defined in the `TileTypes` array, with water being the `defaultTile`. The `Constraints` array includes numerical constraints on cities and a `ConnectivityConstraint` that city, horses and iron tiles are all connected via passable paths. An additional `DistanceConstraint` ensures that cities are at least 7 tiles away from each other (as the `arguments` parameter is `"minimum,7"`) for the map sketch to be feasible. The `Fitness` array includes an `ExplorationFitness` evaluating discovery effort between city tiles, a `resourceSafeAreaFitness` evaluating the extent of passable tiles safe to horse and iron resource tiles with any degree of safety considered (safety threshold of 0.0 in the `arguments`), and how balanced those safe areas are with `resourceSafeAreaBalance`. The maps to be evaluated are stored in an array under

[1]The 4X subgenre of strategy games stands for *"explore, expand, exploit, exterminate"* and revolves around empire building.

```
{"TileTypes":[{"name":"empty","asciiChar":".","p
assable":"true","defaultTile":"true"},{"name":
"wall","asciiChar":"#","passable":"false"},{"name":
"spawnpoint","asciiChar":"P","passable":"true"},
{"name":"healthpack","asciiChar":"h","passable":
"true"},{"name":"weapon","asciiChar":"w","passable":
"true"}],"Constraints":[{"name":"connectAll","type":
"ConnectivityConstraint","referenceTiles":"spawnpo
int,healthpack,weapon"},{"name":"spawnpointNumbe
r","type":"NumericalConstraint","referenceTiles":
"spawnpoint","arguments":"inRange,2,2"},{"name":
"weaponNumber","type":"NumericalConstraint","referen
ceTiles":"weapon","arguments":"inRange,3,5"},{"name":
"healthpackNumber","type":"NumericalConstraint","ref
erenceTiles":"healthpack","arguments":"inRange,6,1
2"}],"Fitness":[{"name":"healthpackSafeArea","type":
"SafeAreaThresholdFitness","referenceTiles":"healthp
ack","arguments":"0.0"},{"name":"healthpackSafeAre
aBalance","type":"SafeAreaThresholdBalance","refere
nceTiles":"healthpack","arguments":"0.0"},{"name":
"spawnpoint+weaponExploration","type":"ExplorationFi
tness","referenceTiles":"spawnpoint,weapon"},{"name":
"spawnpoint+weaponExplorationBalance","type":"Explora
tionBalance","referenceTiles":"spawnpoint,weapon"}],
"Parameters":{"runs":1,"maxGenerations":100,"fi2pop":
"false","population":20,"mutateTogglewall":"5"},"Refe
renceTileMaps":["..#.#..#;P#wh...h;..#.##.#;#.#..#.#;#
....w.#;###.#h#w;....#...;.##...##;hw###h..;..#.#.#.;#
#..#h.#;.#..##..;####.##.;.#....#P;#..#.#h."]}
```

```
["..##...#;P#w....h;..#h##.#;#.#..#.#;#..#.w.#;##..#h
#w;....#...;.##...##;hw###h..;..#.#.#.;##..#.h#;.#..##
..;####.##.;.#...##P;#.#...h."]
```


Table 2: Generation JSON for shooter levels, seeded from an authored level; below is the webservice response.

the `referenceTileMaps` parameter. The response of the `sketchevaluator` method call (at the bottom of Table 3) is an array of two objects (one for each map in the `referenceTileMaps` array): each object has a `scores` parameter, a `feasible` parameter and a `parsedInput` parameter (which denotes which map sketch is being evaluated). Based on the `feasible` parameter, the first map sketch is feasible while the second one is not. Since the first map sketch is feasible, the `scores` object contains the names of fitnesses in the `Fitness` array of the input JSON along with the numerical scores in each. On the other hand, the `scores` object of the infeasible second map sketch contains the names of the constraints in the `Constraints` array of the input JSON along with the numerical scores in each; non-zero constraint scores signify violated constraints, so in this case `DistanceConstraint` and `ConnectivityConstraint` have been violated.

Discussion

As elaborated in the introduction, using a webservice for the purpose of generating map sketches allows novice users to generate game levels without needing to download software or read source code. However, similar functionalities could be offered by a downloadable executable, and there are many


```
{"TileTypes":[{"name":"land","asciiChar":".","passa
ble":"true"},{"name":"water","asciiChar":"~","pass
able":"false","defaultTile":"true"},{"name":"mount
ain","asciiChar":"M","passable":"false"},{"name":
"city","asciiChar":"C","passable":"true"},{"name":
"horses","asciiChar":"h","passable":"true"},{"name":
"iron","asciiChar":"i","passable":"true"}],"Constrai
nts":[{"name":"connectAll","type":"ConnectivityConst
raint","referenceTiles":"city,horses,iron"},{"name":
"cityNumber","type":"NumericalConstraint","referen
ceTiles":"city","arguments":"equals,2"},{"name":
"cityDistance","type":"DistanceConstraint","refere
nceTiles":"city","arguments":"minimum,7"}],"Fitness":
[{"name":"cityExploration","type":"ExplorationFitness"
,"referenceTiles":"city"},{"name":"resourceSafeArea"
,"type":"SafeAreaThresholdFitness","referenceTiles":
"city","targetTiles":"horses,iron","arguments":
"0.0"},{"name":"resourceSafeAreaBalance","type":
"SafeAreaThresholdBalance","referenceTiles":"city"
,"targetTiles":"horses,iron","arguments":"0.0"}],"Ref
erenceTileMaps":["~.~~.~C~;.~.M~..M;~.M....~;Mh..~i~~;
M~hi~.M.;~..M~M~~;..~~...~;~.......;..~....~;M..~~~.~;
.~~~M..~;.~.....~;~~.~M.~~;.~~C~.M~;.M..h..M;.~~M~~~~"
,"~C~~.~.~;....~..M;~.M....~;Mh.C~i~~;M~hi~~M~;~..M~M
~~;...~...~;~.......;..~....~;M..~~~~;.~~~M..~;.~...
..~;~~.~M.~~;.~~.~.M~;.M..h..M;.~~M~~~~"]}
```

```
[{"scores":{"resourceSafeArea":"1.0","cityExplorati
on":"0.9907407407407407","resourceSafeAreaBalance":
"0.25"},"feasible":true,"parsedInput":{"asciiMap":
"~.~~.~C~;.~.M~..M;~.M....~;Mh..~i~~;M~hi~.M.;~..M~M
~~;..~~...~;~.......;..~....~;M..~~~.~;.~~~M..~;.~...
..~;~~.~M.~~;.~~C~.M~;.M..h..M;.~~M~~~~"}},{"scores":
{"connectAll":"0.2857142857142857","cityNumber":
"0.0","cityDistance":"1.0"},"feasible":false,"parse
dInput":{"asciiMap":"~C~~.~.~;....~..M;~.M....~;Mh.C
~i~~;M~hi~~M~;~..M~M~~;...~...~;~.......;..~....~;M.
.~~~~;.~~~M..~;.~.....~;~~.~M.~~;.~~.~.M~;.M..h..M;.
~~M~~~~"}}]
```


Table 3: Evaluation JSON for two strategy game map sketches; below is the webservice response, with the first map sketch being feasible and the second one infeasible.

examples of such generators (e.g. dungeon generators for tabletop RPGs). The main affordances of a webservice (versus a downloadable executable) is its independence of the user's operating system or file execution privileges (which are common concerns with executables) and that the computational burden of artificial evolution is alleviated by moving it to the cloud. This makes level generation possible on mobile devices or web browsers, broadening the adoption potential of the underlying system. Towards this broader appeal, the webservice-based generator is highly customizable and can generate levels for many different genres as demonstrated by the examples included in the paper — admittedly at the cost of specificity which will be discussed below.

An important argument for using webservices rather than open-source solutions is that researchers do not need to document the elaborate (and often obscure and specialized) algorithms used for game content generation. Admittedly, the need for documentation is not bypassed when using webser-

vices; a large portion of this paper consisted of a rundown of the various method calls, parameters and output of the map sketch generation webservice. Allowing more customization to the user of the webservice increases the extent of the documentation, often exponentially; a fine balance must be found between letting users of the webservice the freedom to generate the type of content they need and overwhelming them with required webservice input and with a documentation as extensive as that of the underlying codebase.

An obvious limitation of using a webservice is the need for network access; the map sketch generator can not run offline. Intermittent or limited connectivity can also pose a problem, as the webservice may be unresponsive and the calling software must include a failsafe for such cases. The online nature of the webservice also comes with possible latency issues, either due to a slow connection or due to increased load on the cloud computing service (e.g. when many method calls are being processed). This can pose a problem for real-time level generation; that said, evolutionary algorithms are not ideal for such in the first place.

The webservice-based map sketch generator is expressive and customizable, with the examples in this paper showcasing how levels of several different genres can be evolved or evaluated. However, a limitation of the generator is that it works best with low-resolution abstract levels with few tile types and small map sizes. Moreover, while the numerous integrated playability constraints and fitness functions account for some of the core design patterns of levels, their expressiveness is bounded as the webservice does not allow minute edits at the code level. For instance, the current generator operates on grid-based, top-down views of game levels and the pathfinding algorithm does not handle hex-based maps or side-scrolling game levels. Moreover, the current constraints and fitnesses do not consider the rules or goals of a specific game. For instance, the generated *MiniDungeons* levels do not consider whether a hero can bypass a monster and collect the treasure behind it; this would require a simulated playthrough by an agent adapted for the game at hand (Liapis et al. 2015). The lack of specificity and the small map size conform to the notion of a map *sketch* which provides only the minimal details needed to communicate its purpose (Buxton 2007); these rough sketches can be further refined either by a human designer or by a generator able to use such sketches as input (Liapis and Yannakakis 2015).

Conclusion

This paper motivated the integration of a search-based procedural content generator into a webservice, allowing users to both generate new map sketches and evaluate existing ones — be they generated or hand-crafted. The webservice allows most parameters of the generator to be tailored in the input of the method calls, including the components of the map sketches, their playability constraints and evaluation functions and the parameters of the evolutionary algorithm. Example method calls in the paper showcase how map sketches for 4X strategy games, shooter games, and rogue-like games can be generated from scratch, generated from an initial seed, and evaluated. The self-contained and self-describing nature of this webservice makes it accessible to more users who do not have to be understand (or compile) the underlying code which evolves and evaluates levels, thus reducing the expertise requirements for using the generator.

References

Barros, G., and Togelius, J. 2015. Balanced civilization map generation based on open data. In *Proceedings of the IEEE Congress on Evolutionary Computation*.

Björk, S., and Holopainen, J. 2004. *Patterns in Game Design*. Charles River Media.

Buxton, B. 2007. *Sketching User Experiences: Getting the Design Right and the Right Design*. Morgan Kaufmann.

De Jong, K. A. 2006. *Evolutionary computation - a unified approach*. MIT Press.

Dormans, J., and Bakkes, S. 2011. Generating missions and spaces for adaptable play experiences. *IEEE Transactions on Computational Intelligence and Games* (3):216–228.

Holmgård, C.; Liapis, A.; Togelius, J.; and Yannakakis, G. N. 2014. Generative agents for player decision modeling in games. In *Poster Proceedings of the 9th Conference on the Foundations of Digital Games*.

Kimbrough, S. O.; Koehler, G. J.; Lu, M.; and Wood, D. H. 2008. On a feasible-infeasible two-population (fi-2pop) genetic algorithm for constrained optimization: Distance tracing and no free lunch. *European Journal of Operational Research* 190(2):310–327.

Liapis, A., and Yannakakis, G. N. 2015. Refining the paradigm of sketching in ai-based level design. In *Proceedings of the AAAI Artificial Intelligence for Interactive Digital Entertainment Conference*.

Liapis, A.; Holmgård, C.; Yannakakis, G. N.; and Togelius, J. 2015. Procedural personas as critics for dungeon generation. In *Applications of Evolutionary Computation*, volume 9028, LNCS. Springer.

Liapis, A.; Yannakakis, G. N.; and Togelius, J. 2013a. Sentient sketchbook: Computer-aided game level authoring. In *Proceedings of the 8th Conference on the Foundations of Digital Games*, 213–220.

Liapis, A.; Yannakakis, G. N.; and Togelius, J. 2013b. Towards a generic method of evaluating game levels. In *Proceedings of the AAAI Artificial Intelligence for Interactive Digital Entertainment Conference*.

Lim, C.-U., and Harrell, D. F. 2014. An approach to general videogame evaluation and automatic generation using a description language. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG)*.

Schoenauer, M., and Michalewicz, Z. 1996. Evolutionary computation at the edge of feasibility. In *Proceedings of the 4th Parallel Problem Solving from Nature*, 245–254.

Togelius, J.; Preuss, M.; Beume, N.; Wessing, S.; Hagelbäck, J.; and Yannakakis, G. N. 2010. Multiobjective exploration of the starcraft map space. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*.

Togelius, J.; Shaker, N.; Karakovskiy, S.; and Yannakakis, G. N. 2013. The mario ai championship 2009-2012.

World Wide Web Consortium. 2004. Web services architecture: W3C working group note 11 february 2004. Accessed from `http://www.w3.org/TR/ws-arch/` on July 1st, 2015.

Yannakakis, G. N.; Liapis, A.; and Alexopoulos, C. 2014. Mixed-initiative co-creativity. In *Proceedings of the 9th Conference on the Foundations of Digital Games*.