Taksim: A Constrained Graph Partitioning Framework for Procedural Content Generation

Ahmed M. Abuzuraiq, Arron Ferguson, Philippe Pasquier
School of Interactive Arts and Technology
Simon Fraser University
Surrey, Canada
aabuzura@sfu.ca, arron_ferguson@bcit.ca, pasquier@sfu.ca

***Abstract*—We present Taksim, an Answer Set Programming (ASP) framework for generating content in games through constrained graph partitioning. We illustrate its expressivity by implementing logical constraints that are relevant to generating the spaces of game levels. Furthermore, we present a case study for creating game levels from a given Mission Graph. Finally, we propose key concepts that make constrained graph partitioning, coupled with ASP, an asset for Procedural Content Generation.**

***Index Terms*—answer set programming , procedural content generation , graph partitioning , game level design**

I. Introduction

When generating levels for a game, we want to exert enough control to guarantee the intended experience for the players. The dialogue between the designer's intent and the game spaces that will realize it can be initiated by treating them separately. For example, an intended sequence of tasks to be accomplished by the player can guide the generation of levels [1], [2]. Alternatively, the game spaces can be generated first and validated next against the designer's goals. But starting with either game spaces or design goals may privilege one over the other. Instead, we can start with a discrete underlying structure (such as a Rectangular or a Voronoi grid) and put its cells in groups to form game spaces such as indoor rooms, outdoor regions, or the borders of countries [3]. The process of grouping can then be constrained by high-level design goals while keeping the underlying structure equally malleable and accessible. This way, we can generate both: the underlying structure that will give rise to the game spaces (e.g. Polygonal Map Generation [4]) and the design goals that will guide their formation (e.g. Mission Graphs generation [5]–[7]), provided that the earlier can accommodate the latter.

We see this line of research as an investigation of those ideas through the lens of constrained graph partitioning, expressed in the declarative paradigm of Answer Set Programming (ASP). We outline the framework of Taksim in Section III and illustrate the expressivity of the framework by constructing it in ASP in Section IV. The general concept of partitioning an underlying structure can be used to produce a variety of spatial game spaces such as strategy maps (each partition is a country), dungeons (each is a room) or open-world maps (each is a region) [3]. But in Section V, we demonstrate an important application of Taksim, namely, generating levels that supports a given sequence of in-game tasks (called a Mission Graph). Finally, in Section VI we discuss a number of advantages afforded by the approach taken in Taksim[1].

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

978-1-7281-1884-0/19/$31.00 ©2019 IEEE

II. Related Work

A. Answer Set Programming for PCG

Many design problems are ill-defined, and so it is common for designers to iteratively refine their problem definition [8]. Procedural Content Generation (PCG) is not different in this regard, as the designer of a generator will often find herself building iterations of the generator as means of arriving at a satisfactory design space. The tool used in building and iterating on a generator has to support this process and a case was made by Smith and Mateas [9] for using Answer Set Programming for that purpose.

Answer Set Programming (ASP) is a declarative logic programming variant that is directed toward combinatorial search problems. Problems are encoded in terms of facts, variables, and constraints that relate them. A domain-independent solver can then generate sets of facts that satisfy these relationships and each set is called an Answer Set.

B. Constrained Graph Partitioning

Graph partitioning is the problem of dividing the nodes in a graph to K sets. It is common to enforce constraints on the resulting partitions so they would have uniform sizes [10] or have a certain adjacency relationship to each other [3], [11].

In a previous work by one of the authors [3], it was shown that a number of content generation problems can be seen as instances of the constrained graph partitioning problem. For example, a Delaunay graph which describes the adjacency between Voronoi cells (e.g. as in (1) and (3) in Figure 1) can be partitioned such that each partition represents a different faction in a political map or a different biome in a terrain map. The adjacency between partitions can be described by yet another graph, called the Quotient Graph. Two partitions are considered to be adjacent if any of the nodes belonging to them are adjacent.

The work of Abuzuraiq [3] explores the applications of placing a constraint on which graph partitions should be adjacent or nonadjacent. In other words, creating a partitioning where the Quotient Graph matches a provided Constraint Graph. If the Constraint Graph represented a Mission Graph [2] then the resulting partitions will be a spatial materialization

[1]Taksim means to divide or partition in Arabic, especially when done in a careful and purposeful way.

[2]A Mission Graph is a directed graph describing the sequence of tasks that the player has to accomplish in a level.

of the Mission Graph where each partition can be interpreted as a different room in a dungeon or a different region in an open-world game space.

We carry on the proposal for using constrained graph partitioning for content generation in this work. But instead of using an A* search formulation [3], [12] for addressing constrained graph partitioning, we present a constraint satisfaction approach that is based on Answer Set Programming (ASP) [9]. We found the bar to updating and expressing constraints to be lower in ASP than in the A* formulation. In Section IV, we illustrate that by incrementally constructing a number of constraints that were previously left as future work. Finally, we make a conceptual argument for constrained graph partition in the Discussion.

C. Mapping Mission Graphs to Game Spaces

By separating the generation of Mission Graphs from the generation of spatial game spaces, more control can be afforded to both. A number of approaches for generating Mission Graphs [6], [7] and subsequently converting them to levels are present in the literature. The papers by Dormans [1], [5] are an early work on generating both missions and spaces for action-adventure levels through a combination of graph and shape grammars respectively. Following in spirit is the work by Lavendar [13] for generating levels for the game Legend of Zelda. In a similar line, the work by van der Linden et al. [2] presents an approach for generating dungeons for the game Dwarf Fortress (Wild Card Games 2013) that uses grammars to generate Mission Graphs that are then embedded into a 2D grid through a layout solver. The approach taken in Taksim is to create a game space by partitioning an underlying structure (e.g. a 2D grid or a Voronoi diagram) in a way that is controlled by a set of constraints. The first of these constraints (Adjacency Constraint) is shared with the works mentioned earlier, which is to create regions/rooms that reflect the Mission Graph. For example, if a combat task in the Mission Graph is followed by a puzzle task then we need to have two rooms/regions that serve these purposes and are adjacent to each other.

III. Approach

The core unit underlying Taksim is that of partitions. We obtain these by partitioning an underlying discrete structure through partitioning the graph (named the Basic Graph) that describes the adjacency between its cells (e.g. Voronoi diagram structure and Delaunay graph respectively as in Figure 1). This structure could be a Voronoi, hexagonal or a rectangular grid to name a few. In addition to the Basic Graph, Taksim expects a set of constraints to control the graph partitioning results. For example, we can control the adjacency between the partitions, the length of the borders along which they touch, their sizes and shapes. The underlying assumption is that a variety of game content types can be obtained by exerting the appropriate control over the partitions of a graph, and consequently, the underlying structure they represent. We introduce the overall framework of Taksim and some technical terms next.

A. Technical Terms

- Basic Graph denoted by G is the graph to be partitioned.
- Quotient Graph denoted by Q, is the graph resulting from the partitioning of G, where each node is a partition and an edge is created between two partitions if any of the nodes belonging to them are adjacent in G (e.g. in Figure 1 partitions 4 and 5 are adjacent, but not 4 and 6)
- Adjacency Constraint, determines which the partitions in Q should be adjacent or nonadjacent.
- Constraint Graph, denoted by C, is a graph that represents the desired Adjacency Constraint. If the Adjacency Constraint is satisfied, then Q and C will be connected in the same way. To be brief, we say that the Constraint Graph C is *imposed* on the Basic Graph G.
- Removed Nodes, are the nodes that are removed from the Basic Graph during partitioning in an attempt to satisfy the constraints. For example, if G was a 3×3 grid as in Figure 2 and we wanted to partition it so that Q matches a Constraint Graph which is a cycle of 4 nodes, then we can only satisfy this Adjacency Constraint if we removed the node in the center. Finally, all the Removed Nodes are placed in a special partition to remain consistent with the graph partitioning framework. Throughout this work, we visually represent removed nodes by coloring their respective cells in the underlying structure with a white color (e.g. as in Figure 5).
- Coarsening, which is to produce a smaller version of the Basic Graph G by partitioning it first and using the Quotient Graph of the resulting partitions as a substitute for G. This is exemplified in Figure 1 where the graph in (4) is a coarse version of (2). Coarsening can be used to save computational resources (by reducing the size of the Basic Graph) or introduce variations. We illustrate the latter in Section V.
- Control Constraints, any constraint placed on the partitioning aside from the Adjacency Constraint.

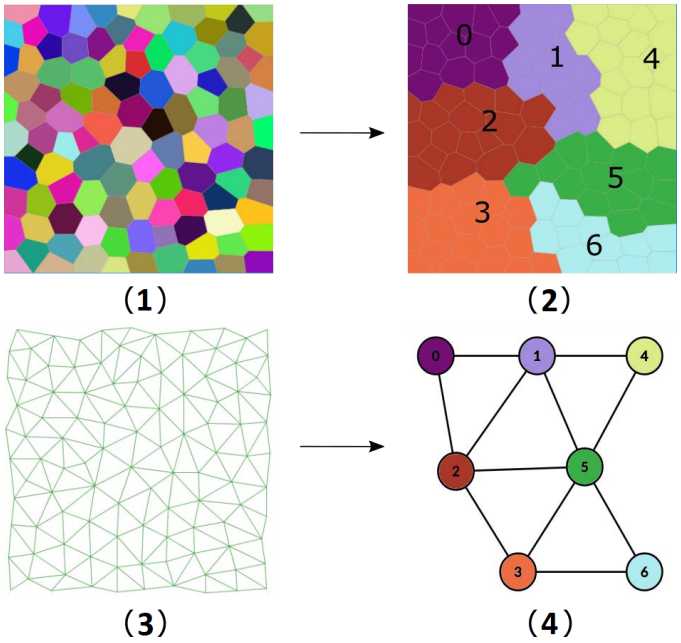

Fig. 1. Reproduced from Abuzuraiq [3]:
(1) A Voronoi grid with Lloyd relaxation.
(2) After partitioning the cells into 7 groups (also an example of coarsening).
(3) The Basic Graph, here it is the dual graph of the Voronoi diagram in (1).
(4) The Quotient Graph of the partitions in (2)

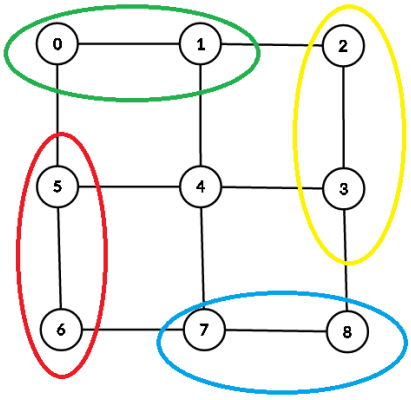

Fig. 2. Reproduced from [3]:
We cannot assign the node in the center to any partition

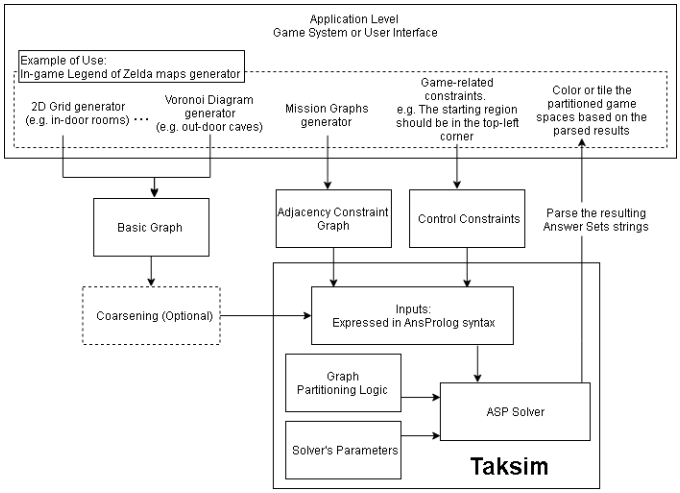

Fig. 3. A system view of Taksim.

B. Framework

Answer Set Programming (ASP) is a declarative logic programming language. The programs in this language are written in the syntax rules of AnsProlog [14] and are then passed to a domain-independent ASP solver. In this work, we used the solver Clingo [15], version 4.5.4 and available at (https://potassco.org/). We will only briefly introduce the main concepts of ASP in the next section, so we refer the readers unfamiliar with ASP to other sources such as Smith's paper [9] or Chapter 8 of the PCG book [16].

Next, we present the overall framework of Taksim from a user's and the system's perspectives (see Figure 3).

1) User: A program or a user interface can be used to generate or hand-design the inputs to Taksim which are the Basic Graph, the Constraint Graph, and the Control Constraints. In order for the ASP solver to reason over these inputs, they must be expressed as AnsProlog statements (e.g. the statements in Section IV-A express the Basic Graph in Figure 4). Finally, the inputs are written to files in preparation to passing them to the ASP solver. Knowledge in ASP is not required when providing these inputs as the system can handle the task of expressing them in AnsProlog, but it is possible that no solution is feasible for a given combination of Basic Graphs and constraints. For example, we cannot *impose* a Constraint Graph that is a cycle of 9 nodes or a chain of 8 on the Basic Graph in Figure 2. This, however, requires an understanding of the used constraints and their relation to the Basic Graph but not necessarily knowledge in the syntax of AnsProlog. The significance of this requirement depends on whether Taksim is used online or offline. In an offline setting, we could mitigate the risk of in-feasibility by using "rich enough" Basic Graphs (In the example above, we can replace the Basic Graph in Figure 2 with that in Figure 1 or Figure 7). In an online setting, a degree of visual feedback and user interaction can provide insights about the feasibility of the constraints and support an interactive redefinition of the inputs and constraints so that they are feasible together.

2) System: The Clingo solver is called through the command line and expects the paths to one or more files containing AnsProlog statements in addition to some solver-specific parameters. Files are divided so that each contains a cohesive unit of AnsProlog statements. The files containing statements for the Basic Graph (Section IV-A) and the main graph partitioning logic (Section IV-B) are always passed to the solver. On the other hand, the files representing the Adjacency or the Control constraints (the rest of Section IV) are only passed if the user provided them from the application level. Finally, the system can be extended by writing ASP statements and passing them to the solver. The explanation we give in Section IV can aid in that.

The result, which will be in a text file, states the partition that each of the nodes will belong to. This is parsed to give it an interpretation. For the purposes of presenting in this paper, we simply color the cells of a Voronoi or a rectangular grid based on which partition they belong to. While Taksim operates on the Basic Graph, it is the structures which the graph represents (e.g. the Voronoi cells) that finally form the geometric layout of a game level or map.

Throughout this system, we uniquely identify nodes in the Basic Graph with integer numbers. A partition can then be defined as a set of numbers that belongs only to that partition. The current implementation assumes that partitions (including the Removed Nodes partition) are mutually exclusive and that they collectively exhaust the nodes of the Basic Graph. Finally, each partition is also assigned a unique integer number.

A dictionary is used to maintain the mapping between the unique number that represents a node and the node object on the application level. For example, a Voronoi cell in Figure 1 is an object containing data about the edges of the cell, its color, and position. The domain-independent ASP solver treats each node as a number only, and the dictionary is then used to map these numbers back to the objects (e.g. Voronoi cells).

Unlike in the previous approach by Abuzuraiq [3], in this formulation, we are not using graph isomorphism as a test for whether the partitions' Adjacency Constraint is satisfied. Instead, we require the edges in the Constraint Graph C and the Quotient Graph Q to match exactly, i.e. if an edge between partitions 1 and 2 exists in C, the same edge with the same nodes numbering should be seen in Q (1-2 and 2-1 are equivalent). In contrast, an isomorphism test will consider two graphs equivalent if they are topologically the same.

TABLE I
A TABLE OF THE TYPE OF CONSTRAINTS. THE SECTION WHERE WE DISCUSS EACH IS SHOWN IN PARENTHESIS.

Relation	Partition-Partition	Partition-Nodes
Constraints	Adjacency (IV-C) Length of Borders (IV-H)	Contiguous Partitions (IV-B) Node Membership (IV-D) Mapping Constraint (IV-E) Nodes Co-Membership (IV-F) Partition's Sizes (IV-G) Internal Graph (IV-I)

IV. CONSTRAINTS IMPLEMENTATION

In this section, we present constraints that can be placed on a graph partitioning and their translation to AnsProlog statements. A complete list of the constraints is shown in Table I where they are categorized into constraints that address the relation between the partitions or their relationship to the nodes in the Basic Graph. The following subsections are to be read sequentially as they build on each other.

A. The Basics

Let's first represent the nodes and edges of the Basic Graph:

```
node(1).
node(2).
node(3).
edge(1,2).
edge(2,3).
```

The above is how the graph in Figure 4 would be represented as facts in AnsProlog. Note how all the statements terminate with dots and that each node is given a unique number that is referenced when edges are defined.

B. Contiguous Partitions

We first show how nodes and partitions are represented and generated, then build few predicates that would aid in defining what constitutes a contiguous partition. So, here are the first few lines:

```
#const n=0.
#const p=0.
#const r=0.
partition(r..p).
1{belongs(N,P):partition(P)}1 :- node(N).
```

The first three lines are constants and their values are passed to the solver using the –const argument. The constant n is the number of nodes in the Basic Graph. The constant p is the number of partitions and it is equal to the number of nodes in the Constraint Graph (if provided). The fourth line represents partitions. The partitions are each assigned a number starting from r and increasing up to p. If node removal is allowed then r is set to 0 and the partition with number 0 will contain all the removed nodes.

The last line contains a choice rule which has the syntax of *n{X:Y}m :- Z* . A choice rule generates a number of facts that are no less than n and no more than m and is the main driver for non-determinism in ASP, in addition to the parameters *rand-freq* and *seed* in Clingo[3]. The last line then translates to: assign each node to exactly one partition, a node is said to belong to a partition using the fact *belongs(N,P)*. Coupled with *rand-freq* argument, this will assign nodes to random partitions but will not guarantee the partition's nodes to be contiguous.

```
family(N1,N2,P) :- belongs(N1,P), belongs(N2,P),
      partition(P), node(N1), node(N2), N1 < N2.
```

Here we define a predicate for whether two nodes belong to the same partition, the last term is to ensure that we always have a single version of the *family* fact in which the node with the lower number comes first. We found that we gain performance improvement by doing so.

```
reach(X,Y) :-
      edge(X,Y),
      family(X,Y,P),
      partition(P).
reach(X,Z) :- reach(X,Y), reach(Y,Z).
```

In the above, we say there is a reach between two nodes if they share an edge and belong to the same partition. We then state that the reach property is transitive. This allows us to decide if two nodes that belong to the same partition have a reach/path between them. We need this test in order to ensure that the partitions will be contiguous. Otherwise, the nodes of the resulting partitions will be scattered.

```
contiguous(P) :- partition(P),
   reach(N1,N2):
   family(N1,N2,P),
   node(N1), node(N2).
:- not contiguous(1..p).
```

In the last line, we use an integrity constraint (has no left-hand-side and reads: if true, then reject) which rejects all the partitions that are not contiguous. We start with the partition with number 1 to exclude the removed nodes partition from this constraint. This integrity constraint narrows down the design space of potential graph partitions. Note how this resembles the generate-and-test pattern.

C. Partitions' Adjacency Constraints

Next, we present how the Adjacency Constraint is implemented. To do that we need to represent the Quotient Graph of the partitioning and then place constraints on it so that it matches the Constraint Graph.

```
edge_q(P1,P2) :-
      partition(P1), partition(P2),
      edge(N1,N2), P1 != P2,
      node(N1), node(N2),
      belongs(N1,P1), belongs(N2,P2).
```

If two nodes in the Basic Graph are connected with an edge and if these two nodes belong to different partitions, then an edge in the Quotient Graph (edge_q) is created between these partitions. The Adjacency Constraints for a graph like in Figure 4 look like this:

```
:- not edge_q(1,2) ; not edge_q(2,1).
:- not edge_q(2,3) ; not edge_q(3,2).
:- edge_q(1,3).
:- edge_q(3,1).
```

The integrity constraints above are a direct representation of the edges in the Constraint Graph. We reject any Quotient Graph that does not have an edge that exists in the Constraint Graph (e.g. :- not edge_q(1,2)). We indicate that either direction of the edge is accepted.

[3]Quoted from the description of Clingo's command line parameters:
–rand-freq= $< p >$: Make random decisions with probability $< p >$
–seed= $< n >$: Set random number generator's seed to $< n >$

Unless we are fine with edges in the Quotient Graph other than the ones we want, we also need to explicitly list the Quotient Graph edges we do not want (e.g. edge_q(1,3)).

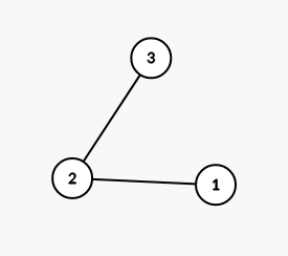

Fig. 4. A chain graph of 3 nodes.

D. Node Membership

We can reuse the *belongs(N,P)* fact and an integrity constraint to reject any solution in which a node does not belong to a chosen partition. The example below ensures that the node with the number 1 will belong to the partition with number 2.

```
:- not belongs(1, 2).
```

E. Mapping Constraint

Given a collection of nodes in the Basic Graph, we can require this collection to be mapped to a certain partition in the result. For example, it could be the case that in a partially hand-designed level, there is a designated region (represented by a collection of nodes) where the player should start or end. In other words, we allow all the partitions to vary (in their shapes and contents) but we expect some of them to be predetermined.

We implement this constraint by replacing the predetermined group of nodes by a single node that represents them. Next, we create edges between this representative node and any node outside the group if there is at least one node inside the group that was linked to it. Next, we assign a unique number to the representative node. This can be the number of any node in the group (since nodes have unique numbers) or any combination of them that is guaranteed to be unique. Finally, we use the Node Membership constraint on the representative node to ensure that it will belong to the desired partition. This process of creating a representative node is done at the application level before running the ASP solver.

Note that this guarantees that the representative node will belong to the sought partition but it will not prevent other nodes to be included in that same partition (i.e. the partition might become larger than intended). To stop the partition from growing, we can restrict its size to be equal to the number of representative nodes in it (1 if only one region was mapped to this partition). We start by defining a predicate for the number of nodes in a partition:

```
count(P,T) :-
      T = #count{N:belongs(N,P), node(N)},
      partition(P).
```

Next, we can constrain the number of nodes like this:

```
:- not count(1,1).
```

Here we want the partition with number 1 to include one node only then due to the Node Membership constraint, it will be the node we formed to represent the region.

F. Nodes Co-Membership

Where we require that two nodes should belong to the same partition (possibly without choosing a one).

```
:- not family(1,4,P):partition(P), P != 0.
```

Here we require nodes numbered 1 and 4 to belong to the same partition. By requiring that *P != 0* we are rejecting cases were they are both added to the removed nodes partition. Combining this constraint with the idea of forming groups of nodes mentioned in the previous Subsection IV-E, we can require that two regions belong to the same partition.

G. Size Optimization

Where we want to maximize or minimize the size of some partitions. This could be motivated by game-specific reasons like maximizing partitions which would represent an ocean or a desert in terrain map or minimizing treasure rooms' sizes, etc. For example, we can use size maximization here. To implement this constraint, we can use the $count()$ predicate we defined earlier along with an optimization statement like this (replace with #minimize for minimization)

```
#maximize {T@1:count(P,T), partition(P), P==1}.
```

This maximizes the size of the partition with number 1, the `@1` indicates the priority of this maximization statement over others. The highest the value, the more priority it is given. We could also maximize all partitions' sizes except the removed nodes partition in a single statement.

```
#maximize {T@1:count(P,T), partition(P), P!=0}.
```

It is important to mention that Clingo accepts a parameter for the number of models to be evaluated, with more models evaluated, the solutions get closer to the optimum in terms of the optimization statements. Passing a value of 0 would request the solver to find the optimal solution by enumerating all models. Since an optimal solution might not be required or desirable, we can terminate the solver after some time or evaluate a limited number of models.

Another way to get partitions with larger sizes is to apply coarsening first on the Basic Graph. This will create partitions with appropriate sizes and more regular shapes (See figs. 5–7, where the same Constraint and Basic graphs were used).

H. Minimizing the Number of Nodes on the Borders between Partitions

One property that we could desire is to limit the number of nodes that are on the borders of partitions. This is important if the partitions are seen as rooms that should connect through gates only (not sharing walls as is often seen in room-based dungeon generators).

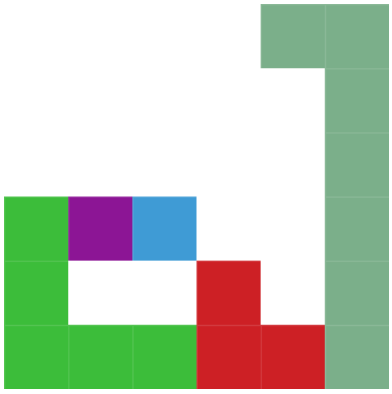

Fig. 5. ASP solver default sample solution. The Constraint Graph is a chain of 5 nodes. White cells are removed nodes.

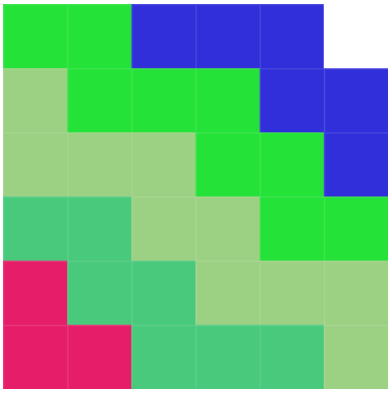

Fig. 6. Here #maximize statements were used with equal priority for all the partitions and 30 models were evaluated, while the sizes are better than in Figure 5, the shapes of the partitions might seem artificial.

This can be achieved through a statement like this:

```
count_borders_supp(P1,P2,T):- T = #count{N1:
      belongs(N1,P1), belongs(N2,P2),
      edge(N1,N2), node(N1), node(N2)},
      partition(P1), partition(P2), P1 != P2,
      P1 != 0, P2 != 0.

count_borders(P1,P2,T) :-
      T = T1+T2,
      count_borders_supp(P1,P2,T1),
      count_borders_supp(P2,P1,T2),
      partition(P1), partition(P2), P1<P2.

#minimize {T@1:
      count_borders(P1,P2,T),
      partition(P1), partition(P2)}.
```

The first line counts the number of nodes on the borders but will give results like *count_borders_supp(1,2,1)* and *count_borders_supp(2,1,3)* for a partitioning like that in Figure 6 (the partition at bottom left has the number 1, one above it has the number 2) that together add up to the right number of nodes at the border so in the second statement we add those together. Note that we don't count nodes on the borders with the removed nodes, but that is only to gain a speed-up and can be omitted. Lastly, we use a minimization statement to reduce the number of nodes at the borders as we discussed in Section IV-G about size optimization.

I. Internal Graph Constraint

For any partition, its internal graph is a sub-graph of the Basic Graph that describes the adjacency between the nodes belonging to that partition. We can control the shape of the internal graph of a partition in a way that is similar to the Adjacency Constraint at Subsection IV-C.

```
is_chain3(P) :-
   node(X1), node(X2), node(X3),
   X1!=X2, X1!=X3, X2!=X3,
   partition(P), belongs(X1,P),
   belongs(X2,P), belongs(X3,P),
   edge(X1,X2), edge(X2,X3),
   count(P,3).
:- not is_chain3(2).
```

Above, we require the partition with number 2 to have the shape of a chain of three nodes (as in Figure 4). We start by stating that we have three nodes, $X1$, $X2$ and $X3$. Next, we make sure that they will be distinct and that they belong to the same partition P. Next, we demand the nodes to be connected in a specific way (1-2 and 2-3). Up to this point, we ensured that at least three nodes in partition 2 will be connected in a chain of three but this does not restrict its internal graph to be only that. This is why we also restrict the size of the partition through the *count(P,3)* term. Remaining is to note that the above constraint can be automatically constructed on the application level in a way that adapts to any internal graph provided by the user. Below, is an additional example for an internal graph that is a cycle of four nodes:

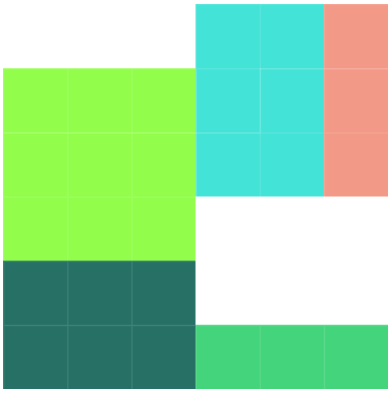

Fig. 7. Here a coarse version of the Basic Graph is created first. The blocks created through coarsening sets a limit on the minimal partition size, also the partitions' shapes are influenced by the blocks' shapes which are in turn controllable by changing the algorithm used for coarsening.

```
is_cycle4(P) :- partition(P),
   node(X1), node(X2), node(X3), node(X4),
   X1!=X2, X1!=X3, X1!=X4,
   X2!=X3, X2!=X4, X3!=X4,
   belongs(X1,P), belongs(X2,P),
   belongs(X3,P), belongs(X4,P),
   edge(X1,X2), edge(X2,X3),
   edge(X3,X4), edge(X1,X4),
   count(P,4).
```

Figure 8 shows the above constraint in use.

V. Showcase: Legend of Zelda

In this section we convert the Mission Graph in Figure 9 to an open-world level. Figure 8 shows a partitioning of a 6×6 grid. An internal graph constraint (is_cycle4) is applied to the boss and start regions while a size constraint that restricts the size to a single node is applied to the boss key, small key, and dungeon item partitions. Moreover, the remaining partitions (combat and puzzle) had their sizes restricted between 3 and 6 nodes and the number of nodes on the borders between all partitions is minimized with 10 models evaluated for that purpose. Finally, We placed the icons for the locks and keys from the online series by Brown [17] to illustrate how they relate to the Mission Graph in Figure 9.

While starting with a small grid is a good strategy for iterating quickly on the constraints and getting fast feedback, it is often more interesting to generate levels at a larger scale. In Figure 10 the Basic Graph is grid of size 14×14. A coarse version of the Basic Graph is created that has 36 nodes only (instead of 196). The same constraints are used here but the appearance of the partitions here is affected by the shape and size of the blocks created by the coarsening process (showing coarsening as a source of variation). We note here that the Mission Graph can contain cycles and consequently, the generated levels that satisfies them. An advantage of that is that it reduces dead-ends and player's backtracking [18]. For an example, see Figure 10.

VI. Discussion

Starting with an underlying structure as a substrate, and constructing content by partitioning it, we gain these advantages:

A. Separation of Concerns

First, we largely separate the concern of content generation which is performed in an abstract constraint satisfaction manner (e.g. on the Delaunay graph), from the aesthetics and geometry of the content which stem from that which the Basic Graph represents. For example, cells in a Voronoi diagram can be rendered after applying noise to their edges and coloring them. Amit Patel's article on Polygonal map Generation shows a good example of that [4].

B. Diverse Sources of Variation

The variation in the content can be introduced by randomizing the Basic Graph, the Constraint Graph or the Control Constraints. Another strategy is to start with a large Basic Graph and coarsen it first (e.g. as in Figure 1) to produce nodes with more random shapes as we did in the last section to generate the level in Figure 10. Even if the above could not be varied, then there are many ways to partition a Basic Graph given a fixed set of constraints. This is made true through choice rules in ASP and parameters such as *rand-freq* and *seed* in Clingo.

C. It is Graphs All the Way Down

When nodes are partitioned into groups, a new Quotient graph is created. This resulting graph can then be treated as a Basic Graph and be partitioned subject to a different set of constraints. This allows us to partition a graph then group the partitions into larger ones and so on. This could be used to create a political map recursively, where we start with forming small states then further group them into provinces and finally

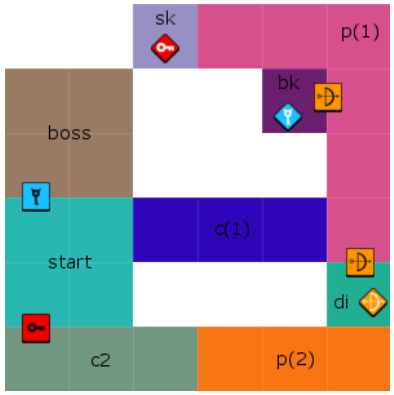

Fig. 8. The Constraint Graph is shown in Figure 9. The internal constraint is_cycle4() is applied to the boss and start nodes.

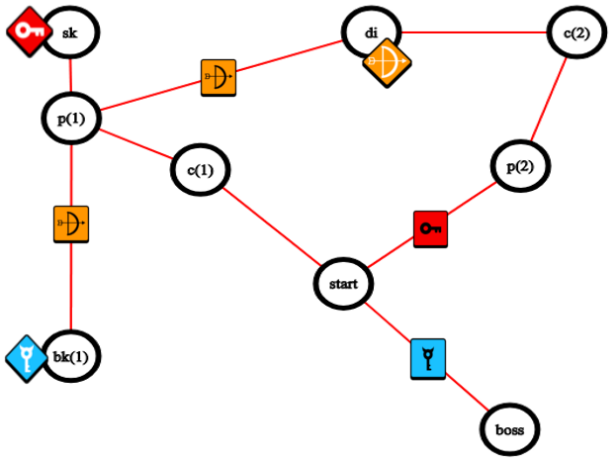

Fig. 9. We follow the naming convention of Smith et al. [7] in this Mission Graph.
sk: Small Key, *di*: Dungeon Item,*bk*: Boss Key, *c(1)* and *c(2)*: Combat, *p(1)* and *p(2)*: Puzzle. Remaining are *sl*: Small Lock, *dl*: Dungeon Lock and *bl*: Boss Lock. For these we decided to annotate them on the Constraint Graph's edges instead of giving them a separate node.

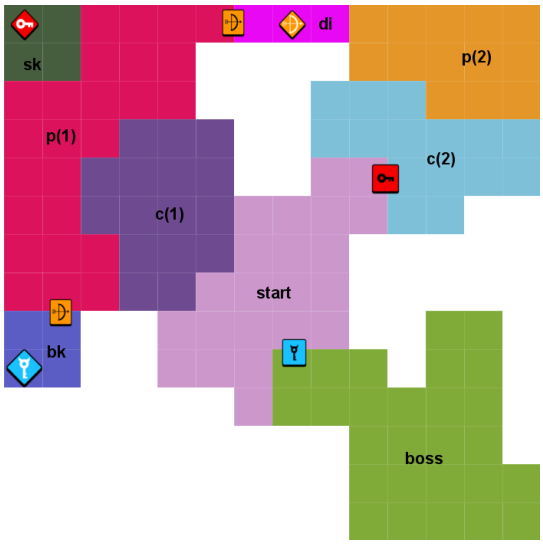

Fig. 10. An open-world level. Since the Mission Graph contains a cycle, the player does not need to backtrack through ***c(2)*** once arriving to ***di***. Instead, they can complete a full cycle by proceeding to ***p(1)***.

group the provinces to form countries. We can also do it in the opposite direction by starting with partitioning the Basic Graph to form countries and then given the internal graph of each country, we partition again.

D. Access to Data

The third advantage can be highlighted by contrasting our approach to the problem of converting a Mission Graph to a game space (dungeon rooms, open-world regions, ...etc) with other approaches. In particular, we refer to the work by Dormans and van der Linden [2], [5] as top-down approaches since they consider the generation of the game space after obtaining a Mission Graph. Another (more common in practice) class of approaches generates game spaces and leaves the Mission Graph as an emergent and implied property of the generation, we refer to these as bottom-up approaches. Constrained graph partitioning can be seen as a combination of the two since both the Basic Graph and the Mission Graph can be generated separately, for example through approaches such as Polygonal Map Generation [4] and generating Mission Graphs with lock-and-key structures [7] respectively. The data associated with the structure underlying the Basic Graph is available prior to and during partitioning. This allows us to access the data associated with the partitioned nodes and express Control Constraints in their terms. This data could be the partition they belong to (hence being able to express the Node Membership constraint), but it could also be their position, geometry or other domain-dependent semantics such as terrain, or collectibles. Since Taksim uses Answer Set Programming, we can perform the partitioning while taking this data into consideration as facts in AnsProlog.

E. Integration with Non-Procedural Elements

Finally, the fourth advantage concerns the fact that the Basic Graph does not need to be generated. Instead, it can be the navigation mesh of a hand-designed level or any graph structure as long as the act of partitioning it has a meaning. If we combine this invariance to generation with the use of the Mapping Constraint (Subsection IV-E) we argue that Taksim is capable of integrating procedural elements into games that contains a hybrid of both generated and hand-designed content.

VII. Conclusions and Future Work

We proposed Taksim, which is an ASP-based constrained graph partitioning framework for content generation. We gave an overview of the framework and its implementation, illustrated a result of converting a Mission Graph to game spaces, and argued conceptually for the advantages of the approach.

In the future, we can explore capitalizing on the point made in VI-D by adding properties to the nodes and partitions (e.g. node(1, DESERT) or partition(3, BOSS_ROOM)) in a way that makes it more customizable to the domain at hand. For example, demanding a certain percentage of steppes tiles in the partition/country representing a nomadic tribe.

Currently, we only allow removing nodes from the Basic Graph. Upon observing Lavendar's [13] implementation of Dorman's [1] shape grammars, we find that it satisfies the Adjacency Constraint by removing or creating walls between the rooms in a level. We note that this is one way for representing the absence or presence of an edge in the Basic Graph, visually and geometrically. As a result, we are interested in exploring the potential of allowing edge removal in Taksim.

Another venue we want to explore is the use of a mixed-initiative interface to enable the visual expression of constraints and an easier iteration on them. The designer can interactively select subgraphs of the Basic Graph to partition with different constraints for each. The tool would capitalize on the human's ability to detect visual patterns in a way that makes the partitioning guaranteed to find (e.g. by interactively removing nodes from the Basic Graph) and its results closer to the designer's intent. Finally, we plan to continue exploring the applications of Taksim to different types of content in games. In addition to that, we will consider game levels of a larger scale than presented here, possibly through the support of the aforementioned mixed-initiative interface.

References

[1] J. Dormans, "Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, ser. PCGames '10. New York, NY, USA: ACM, 2010, pp. 1:1—1:8. [Online]. Available: http://doi.acm.org/10.1145/1814256.1814257

[2] R. der Linden, R. Lopes, and R. Bidarra, "Designing procedurally generated levels," in *Proceedings of the the second workshop on Artificial Intelligence in the Game Design Process*, 2013.

[3] A. M. Abuzuraiq, "On using graph partitioning with isomorphism constraint in procedural content generation," in *Proceedings of the 12th International Conference on the Foundations of Digital Games*. ACM, 2017, p. 77.

[4] A. Patel, "Polygonal Map Generation for Games," p. 4, 2010. [Online]. Available: www-cs-students.stanford.edu/ amitp/game-programming/polygon-map-generation/

[5] J. Dormans, "Level design as model transformation: a strategy for automated content generation," in *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*. ACM, 2011, p. 2.

[6] D. Karavolos, A. Liapis, and G. N. Yannakakis, "Evolving missions to create game spaces," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2016, pp. 1–8.

[7] T. Smith, J. Padget, and A. Vidler, "Graph-based Generation of Action-adventure Dungeon Levels Using Answer Set Programming," in *Proceedings of the 13th International Conference on the Foundations of Digital Games*, ser. FDG '18. ACM, 2018, pp. 52:1—52:10. [Online]. Available: http://doi.acm.org/10.1145/3235765.3235817

[8] J. C. Thomas and J. M. Carroll, "The psychological study of design," *Design studies*, vol. 1, no. 1, pp. 5–11, 1979. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0142694X79900206

[9] A. M. Smith and M. Mateas, "Answer Set Programming for Procedural Content Generation: A Design Space Approach," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 187–200, 2011.

[10] G. Karypis and V. Kumar, "Multilevel Algorithms for Multi-constraint Graph Partitioning," in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, ser. SC '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 1–13. [Online]. Available: http://dl.acm.org/citation.cfm?id=509058.509086

[11] O. Moreira, M. Popp, and C. Schulz, "Evolutionary Acyclic Graph Partitioning," *arXiv preprint arXiv:1709.08563*, 2017. [Online]. Available: http://arxiv.org/abs/1709.08563

[12] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[13] B. Lavender and T. Thompson, "The Zelda Dungeon Generator: Adopting Generative Grammars to Create Levels for Action-Adventure Games," 2015.

[14] C. Baral, *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[15] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "Clingo = ASP + control: Preliminary report," *CoRR*, vol. abs/1405.3694, 2014.

[16] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural Content Generation in Games*. Switzerland: Springer International Publishing, 2016.

[17] M. Brown, "How my Boss Key dungeon graphs work — {Game Maker's Toolkit} on {Patreon}," aug 2017. [Online]. Available: https://www.patreon.com/posts/how-my-boss-key-13801754

[18] J. Dormans, "Cyclic Generation," in *Procedural Generation in Game Design*. AK Peters/CRC Press, 2017, pp. 83–96.