On Using Graph Partitioning with Isomorphism Constraint in Procedural Content Generation

Ahmed M. Abuzuraiq
Information and Computer Science Department
King Fahd University of Petroleum & Minerals, Saudi Arabia
abuzreq@gmail.com

ABSTRACT

This paper describes an algorithm to solve the problem of partitioning a planar graph with a constraint on which partitions should be adjacent or nonadjacent. We explore the applications of the algorithm in Procedural Content Generation in games which includes: the generation of political maps, distribution of terrain and converting or linking Mission Graphs to game spaces. We solve this problem using A-Star search with a heuristic for measuring graphs similarity and we suggest techniques such as graph coarsening to limit the search space. The algorithm sensitivity to the initial state is analyzed next and a restart policy is suggested to overcome that. Additionally, we present multiple constraints that can aid in better controlling the outcomes of the algorithm and we show how these constraints can help in the implementation of the displayed applications.

CCS CONCEPTS

• **Applied computing → Computer games**; • **Computing methodologies → Discrete space search**; • **Mathematics of computing** → *Spectra of graphs*;

KEYWORDS

Games, Procedural Content Generation, A-Star Search, Graph Isomorphism, Graph Partitioning, Quotient Graph, Isospectrality, Graph Coarsening, Political Maps, Mission Graph, Restart Policy

ACM Reference format:
Ahmed M. Abuzuraiq. 2017. On Using Graph Partitioning with Isomorphism Constraint in Procedural Content Generation. In *Proceedings of FDG'17, Hyannis, MA, USA, August 14-17, 2017,* 10 pages.
https://doi.org/10.1145/3102071.3110575

1 INTRODUCTION

Procedural Content Generation is the algorithmic creation of digital content. Such content includes levels, game spaces like dungeons, caves, maps and planets. It may also include assets like music, textures, narrative and stories or the game rules themselves. The attractiveness found in these methods, other than the mere joy of creating something out of nothing but code, is that they may reduce the costs of development and make the game more replayable[3]. They also allow the game to adapt for the player through its content and sometimes the content generated may be surprising because it was not imagined that a human designer might have designed the same content manually. In this paper, we describe an algorithm that was inspired by the previously published work by Amit Patel [17] and was extended by considering some important constraints that enable developers to have better control over the final outcomes of their content generators. This paper is organized as follows: Section 2 surveys some related work, followed by the problem formulation in Section 3. Section 4 describes our proposed algorithm. Section 5 outlines the obtained results followed by a showcase of examples in Section 6. We give detailed applications of this algorithm in Section 7, and finally giving our conclusion and future work in Section 8.

ACM ISBN 978-1-4503-5319-9/17/08…$15.00
https://doi.org/10.1145/3102071.3110575

2 RELATED WORK

Patel's article *Polygonal Map Generation for Games* [17] had been a source of inspiration for many other generators (e.g. [2, 15]) including our work. One of the important ideas he used was to base generation of maps on a Voronoi diagram. The advantage of doing so is that the generated content will have a graph and polygonal structure with a more organic shape than when rectangular or hexagonal grids, for example, are used. Also, the cells of the Voronoi diagram can be assigned different properties of terrain, quests, narrative, monsters, etc. We sought to utilize this idea in political maps[1] generation but wanted to add an important constraint that is specific to political maps, which is to allow the designer to provide the graph that describes the adjacency between the countries/regions of the map.

The importance of this constraint appears, for example, in that the number of edges coming into or out of a region impacts how easy/hard it is to attack/defend, the number of routes that can go through it, the spread of disease, etc.

We do this through partitioning the dual graph of that Voronoi diagram (see Figure 1) or any planar graph with a constraint on which partitions should be adjacent/in-adjacent. In other words, how should the quotient graph[2] look like. We were not able to find a paper on graph partitioning, according to the best of our knowledge, that imposes the same constraint that we require. A closely-related paper was by Moreira et al. [13] but in it the authors only required that the quotient graph is directed and acyclic (DAG).

[1]Political maps are a representation of the borders between countries, regions, provinces, etc.
[2] The quotient graph is a graph in which every node corresponds to a partition block. Two partition blocks A and B are said to be adjacent if at least one node in A is adjacent to one node in B.

Other than that, the constraints usually discussed are: minimizing the number of edges between the partitions and balancing the partitions sizes [10] and requesting that certain nodes in a graph should belong to certain partitions of that graph [23].

One strategy of generating game levels is described by Joris Dormans [4]. In this paper, the author suggests that we should see a game level as being composed of two structures: the missions/tasks which the player is going to do and the space in which they will take place. We then generate the missions and create a space after that which suits the generated missions. The missions are represented as a Mission Graph which is a directed graph where an edge destination is a task that is made available when the task at the edge source is accomplished. In Section 7.9 we will present our algorithm as a method of converting a Mission Graph into a game space. We formally specify the problem next.

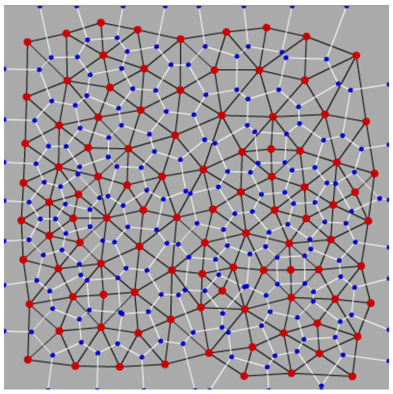

Figure 1: The blue dots and the white edges form the Voronoi diagram. The red dots and the black edges form its dual graph (Delaunay triangulation). [17]

3 PROBLEM FORMULATION

We describe the problem we attempt to solve as follows:

Given an undirected planar graph $G = (V, E)$, partition G into k contiguous partitions such that the quotient graph $Q = (v, e)$ is isomorphic to a given graph $C = (v', e')$. Two graphs are said to be isomorphic if they have an equal number of nodes that are connected in the same way.

In Section 7, we will show what applications will a solution to this problem have in procedural content generation. To be brief we will use the phrase "C is imposed on G" to mean C is isomorphic to a quotient graph Q of G. The basic graph G which we will use for illustration is the dual of the Voronoi diagram which is planar, connected and simple (one edge per two nodes at most). Also, because G is planar and we require the partitions to be contiguous, C must also be planar.

4 APPROACH

4.1 A-Star Search

To solve the problem described earlier, we used A-Star search algorithm [1, 7, 18]. Given a current state s we find the next states that result from applying all the possible actions at s and we pick the one with the lowest cost. The cost is computed as follows: $cost(s) = g(s) + h(s)$ where $g(s)$ is the uniform cost of the state s which is the number of actions that led to this state and $h(s)$ is the value of the heuristic. The heuristic is an estimate of the cost of the path from a state to the solution which if chosen right can lead to pruning large parts of the search tree.

4.2 Initial State

Every state in the search tree is a possible partitioning of G (i.e. a quotient graph). We obtain the initial state by partitioning G into $|C|$[3] partitions through Kernel Clustering [24][21] algorithm which we found to result in a somewhat balanced partition sizes reducing the sensitivity to the initial state. This algorithm is deterministic and so variations are introduced by using different graphs G whereas in Section 7.10 we will describe another randomized algorithm to obtain different initial states from the same graph G.

4.3 Goal Testing

Testing for the goal state is done through VF2 [16] algorithm for graph isomorphism determination. It is known that the problem of graph isomorphism, when restricted to planar graphs, is solvable in polynomial time [20].

4.4 Possible Actions

At any state, which is a possible quotient graph of G as we mentioned, the actions possible per partition are taking a neighboring node from an adjacent partition into its members (node-label change) or removing a neighboring node from the graph G. For example, in Figure 3 at the top, the actions possible are adding the nodes 0,1,2,3,5,6 or 8 to the blue partition (which includes 4 and 7) and similarly for the other partitions. We restrict the choice for each partition to the neighboring nodes because we only want contiguous partitions. For the same reason, we also made sure that taking a node from another partition will not disconnect that partition (which will rule out the nodes 3 and 5 in the previous example). Finally, we also prevent a node to be taken from a partition if that was the only node left in it.
As for node removal, we find this action to be very helpful in the cases where the choice of C is hard or impossible to impose on G, for example, it is impossible to impose the graph C_4 (a cycle of four nodes) into the graph that is a grid of size 3×3 unless we remove the node in the middle of the grid (See Figure 2 and Figure 13). Node removal is also necessary when C is disconnected and G is connected (See Figure 8). In addition to that, applying the removal action will reduce the number of actions that we need to consider in future states.
The resulting state of each action is the quotient graph of G after applying the action. (See Figure 3)

[3]|C| means the number of nodes in the graph C.

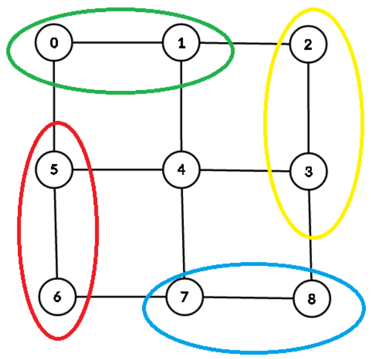

Figure 2: We cannot assign node 4 to any partition

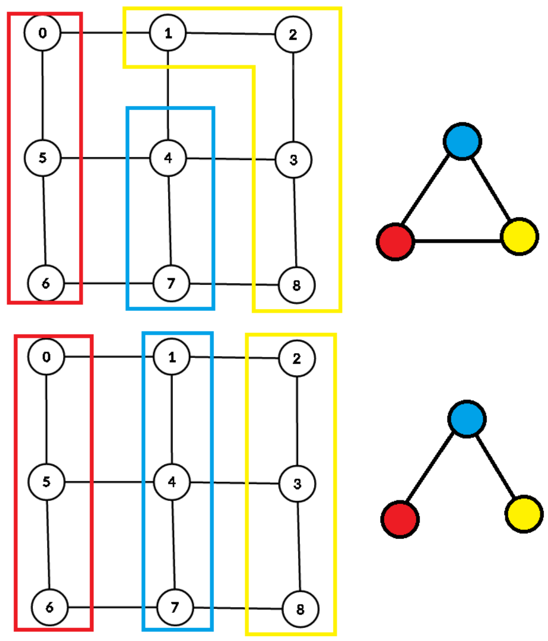

Figure 3: (Above) Quotient graph before action, (Below) Quotient graph after the action of adding node with number 1 to the blue partition.

4.5 The Actual Cost

Every action on the level of the basic graph G, whether it was changing the partition to which a node belong or removing it from the graph, will correspond to an edge removal or addition on the level of the quotient graph Q. This means that the actual cost of the path between the current-state/quotient-graph and goal-state/constraint-graph is the minimum graph edit distance between them. Computing it, however, is NP-Hard [25].

4.6 The Eigenvalue Heuristic

Instead of using the graph edit distance, we use an $O(n^3)$ time complexity heuristic that is based on the difference between the eigenvalues of the Laplacian of the current-state quotient graph and the constraint graph. The heuristic relies on the fact that every two isomorphic graphs are also isospectral (i.e. have the same eigenvalues for their Adjacency or Laplacian matrix). So as we get closer to the goal we expect that the current quotient graph will become closer to being isospectral with the constraint graph. Namely, we use the Eigenvalue Method of computing graph similarity[4] described in the work of Koutra et al. [9, 11].

The heuristic then becomes the similarity between the current quotient graph Q and the wanted constraint graph C. Where the similarity using this method takes values in the range $[0, \infty)$ with a value of 0 meaning that the two graphs are completely similar (The edit distance between them is 0) or increasingly dissimilar as the value moves further from 0.

Not every two isospectral graphs are isomorphic and this was the reason we are using the A-Star search instead of the greedy Best-First search, i.e. to avoid being stuck at a local minimum at which the quotient graph Q found is isospectral but not isomorphic to C.

For A-Star search to be optimal, i.e. guaranteed to find a solution, the heuristic used must be admissible, meaning that its value is always less than the actual cost from that evaluated state to the final state.

In our case, the heuristic is, in general, monotonically decreasing but with the values of the heuristic not always less than the actual cost.

4.7 Coarsening

4.7.1 Main Idea. When the size of the basic graph G increases, the number of actions will also increase. To reduce the number of actions we can pre-process the graph G by putting its nodes into groups and then treating every group as one node. More formally, what we do is that we partition G into k partitions and the quotient graph will then be the new G which we will call G'. The same algorithm that we used to obtain the initial quotient graph can be used here (See Figure 4). The actions this way become the removal of a group of nodes or moving them between partitions.

4.7.2 On the Number of Groups to be Chosen. The value of k has to be well chosen. It cannot be, for example less than or equal to $|C|$ and cannot be very close to the value of $|C|$ so as to leave some room for actions (in other words, a large enough search space for a solution to be found). e.g. if $k = |C| + 1$ then upon finding the initial state, every partition will have one node from G' except for one which will have 2 nodes. What this means is that only one node can be moved between partitions or removed from G' at any time which might mean, in some cases, that a solution cannot be found. The value of k should not be too close to $|G|$ as well since this would defy the point of coarsening in the first place. Therefore, we want to choose k that is not too small that it makes the problem unsolvable and not too large that we solve the problem at a higher size of G' than is necessary.

4.7.3 Incremental Coarsening. One idea we used was to try different values of k incrementally. We first start with a value close to C then if a solution was not found[5] or a maximum number of states were explored we increase k by d. Where choosing d to be as

[4]The more similar two graphs are, the smaller is the edit distance between them.
[5]If G' is small, then the search space will be small and this can be known quickly.

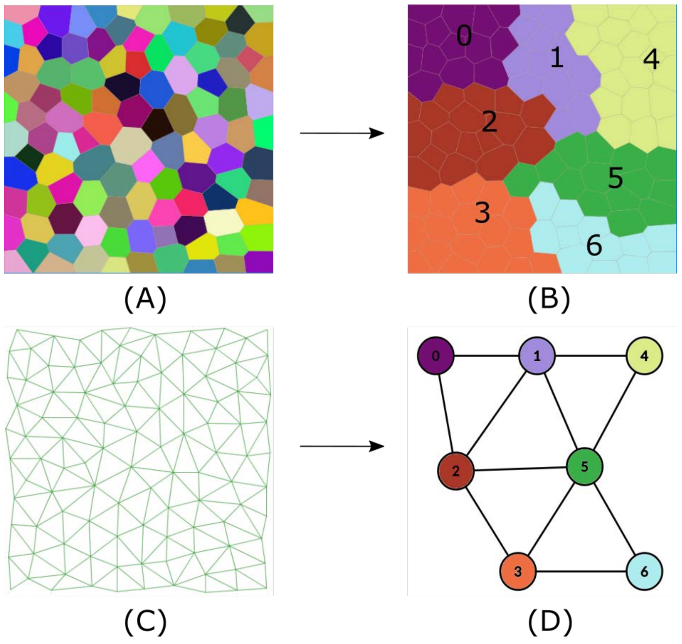

Figure 4: (A) The Voronoi diagram, (B) After putting the cells into 7 groups. (i.e. $k = 7$), (C) The dual graph of the Voronoi diagram in A, (D) The quotient graph of the groups in B.

small as 1 or 2 means that the current coarsening will not be much different from the next coarsening and therefore we cannot expect much difference in the output either. d must then be large enough to avoid that but not too large so as not to solve the problem in a higher dimension when it is more easily solvable in a lower one.

We continue these increments until the value of k becomes equal to or larger than $|G|$.

This idea is influenced by the coarsening step found in the Multilevel algorithm for graph partitioning [8] and its main advantage is that it will allow us to handle large graphs as we handle smaller ones with the natural differences in the amount of memory used and the computations needed for the coarsening.

5 RESULTS

The worst case complexity of A-Star algorithm is $O(b^d)$ where d is the length of the path to the solution and b is the branching factor which is the average number of actions per state. The actions per state is the collection of actions per partition of both neighboring nodes removal and addition to the partition. In the best case, a perfect heuristic will lead directly to the solution with a complexity of $O(d)$

All the tables in this section share the following configurations: Constraint graph C in Figure 5. Each graph G is a dual graph of a Voronoi diagram like the graph in Figure 4. Each run with a different random graph G. Machine Specifications: Processor Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz.

5.1 Sensitivity to the Initial State

We found that the efficiency of the algorithm is reliant on the initial partitioning which, in turn, is reliant on the graph G since the algorithm through which we partition G is deterministic[6]. To verify this sensitivity to the initial state we fixed C and generated G with different seeds. Table 1 shows the results of this test. The graph C that we chose is the chain of 8 nodes seen in Figure 5. Looking at the Time column, we observe that while the STD and the maximum are high in the two columns, the median is low.

In many combinatorial search problems, it is observed that when a deterministic search algorithm is applied on several random instances of a problem (or equivalently several initial states) the algorithm runs exhibit a heavy-tailed distribution. This means that any run has a small probability of taking exponentially longer time. To solve this problem, we employed a restart policy as suggested by Gomes et al. [6].

The restart policy we used is as follows: a small limit L is chosen first, if the search exceeds this limit then the run is stopped, the limit is increased by an amount D and the search is started again with a new random instance (i.e. a new graph G). The intuition is that the computational cost of a sequence of short runs will be smaller than that of a single long run. We chose to set the limit on the number of expanded nodes.

As suggested by Table 2, this made the algorithm less sensitive to the initial state by bringing the times taken by the algorithm runs closer to the median. The median increased due to the wasted nodes expansions and time taken in the runs in which a limit was exceeded.

Table 1: Testing sensitivity to initial state. Data obtained from 100 runs.

	Expanded Nodes	Path Length	Time (Secs)
Max	18386.00	11.00	163.83
Min	3.00	3.00	0.30
STD	1989.56	1.48	17.40
Avg	466.76	6.77	4.83
Median	66.00	7.00	1.34

Table 2: After using a Restart Policy, Initial Limit = 10 ,Limit Increment = 50. Data obtained from 200 runs.

	Expanded Nodes (Last Run\Wasted\Total)	Path Length	Total Time (Secs)
Max	179.00\350.00\437.00	12.00	43.46
Min	4.00\0.00\4.00	4.00	0.29
STD	34.70\65.00\88.10	1.27	5.47
Avg	34.15\92.25\126.40	6.48	5.29
Median	21.00\100.00\106.00	6.00	3.48

[6]i.e. Kernel Clustering algorithm in [24]

5.2 Effect of Coarsening

As shown in Table 3, coarsening shortens the time needed by the algorithm. The Restart Policy from Section 5.1 is used, Initial Limit = 10, Limit Increment = 100. Note that the times shown in the table include the time wasted in runs in which a limit was exceeded.

Table 3: Effect of coarsening. Data obtained from 100 runs. k is the number of groups.

	Time		
	No Coarsening	**$k = 30$**	**$k = 20$**
Max	38.68	38.45	24.45
Min	0.61	0.32	0.20
STD	8.04	6.86	3.73
Avg	7.51	5.46	3.38
Median	4.68	3.00	2.25

5.3 Handling Inefficiency

We found it challenging to state concisely when does the constraint graph C become hard to impose on G. We hope to explore this more in the future. But in general, addition of edges to C can increase its hardness especially if the nodes to which the edges were added had a high degree. A constraint graph is especially hard to impose if it's maximal planar[7].
Building on what we discussed in Section 4.7.2, the size of G' (or G in general) should be large enough to provide the required freedom of actions. But also, increasing the size of G will increase the size of the search space. Still, it's inevitable to increase the size of G or G' when the size of the constraint graph and its topology requires that. In these cases, some of the following techniques can be used.

5.3.1 Accepting Approximations. In applications where the isomorphism constraint can be softened, we can impose disjoint subgraphs of the constraint graph C on disjoint subgraphs of the basic graph G. The result will be correct within every subgraph of G but the adjacencies between the partitions that don't belong to the same subgraph of G might not match our initial constraint graph.

5.3.2 Online PCG. If the content is generated while the game is running, as opposed to during its development, the PCG is called Online [19]. The inefficiency we mentioned makes it challenging to use this algorithm in Online PCG without some intervention from the developers like only allowing the constraint graphs that we know, before hand, will be easier to impose on C and possibly also providing the seeds for the initial states that we found to be suitable at the time of development (Offline).

5.3.3 Mixed Initiative PCG. Mixed Initiative PCG is a term that applies to content generation tools or algorithms in which both the generator and the human using it co-operate to create the content [12]. It's possible to create a Mixed Initiative tool through which the user (developer or player, based on ease of use and learning) can generate or import an embedded planar graph G (it has to be embedded for the user to be able to see it and interact with it), choose a proper subgraph of the graph G and impose a constraint graph on it, then repeat that for the other subgraphs aided with the ability to remove nodes from G in a way that supports the constraint imposing. Additionally, the tool might provide the user with the option to pause the search at any time, then the best state/s (as evaluated by the heuristic) is shown and the user can suggest the next actions to be taken. The goal is to utilize the human ability to find visual patterns. Also, through such a tool, the additional constraints which we will discuss in Section 7 can be more easily (and visually) defined. The main advantage, as we see it, will be that the user will care mainly about the end result regardless of what means were used to obtain it.

In addition to the above, in Section 7 we will introduce the idea of utilizing the recursive nature of some problems in the context of political maps.

[7]A maximal planar graph has the property that adding any more edges to it will result in a nonplanar graph.

6 SHOWCASE

In each showcase the constraint graph C is on the right and the resulting partitioning on a Voronoi diagram graph is on the left. White cells represent removed nodes.

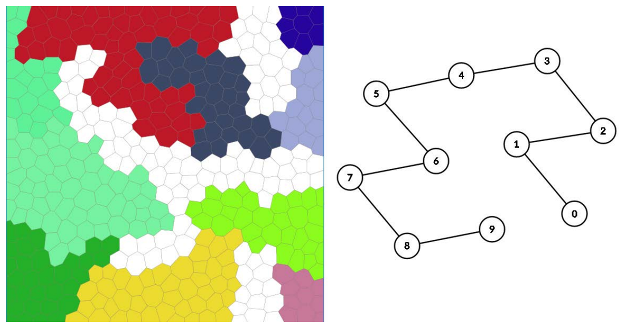

Figure 5: Showcase 1

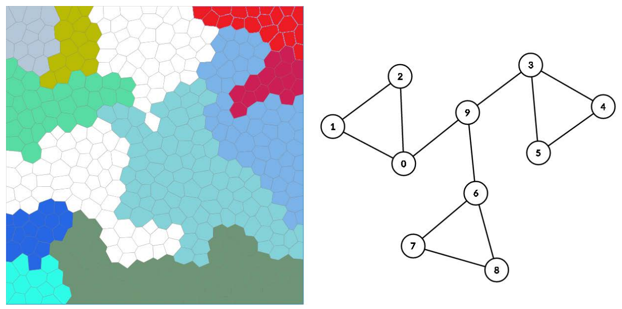

Figure 6: Showcase 2

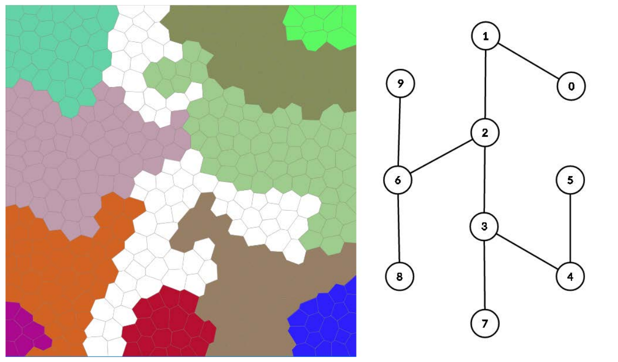

Figure 7: Showcase 3

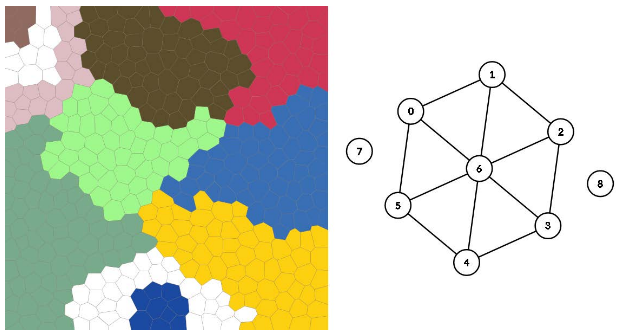

Figure 8: Showcase 4

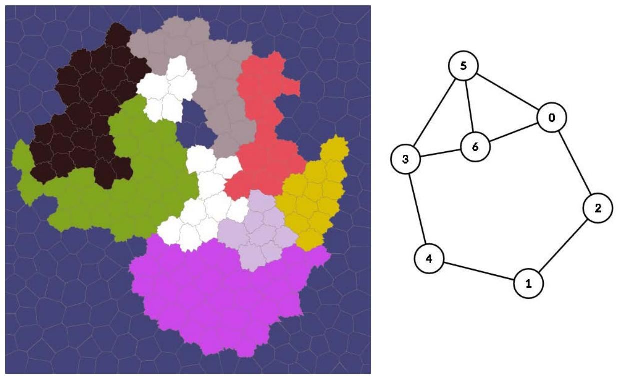

Figure 9: Showcase 5. G is the interior of the island. The blue cells (water) are not part of G.

7 APPLICATIONS

7.1 General Requirements

No matter what the application is, the following are needed:

- A graph G on which the partitioning to be applied.
- A graph C which describes the required adjacency between the partitions.
- (If coarsening is applied) The value for k and if incremental coarsening is used the increment d.
- (If the Restart Policy is used) The initial limit L and the limit increment D.
- A goal test, given a state returns whether it's a solution.
- A heuristic.
- Whether node removal is allowed.
- Any additional constraints (see next).

7.2 Using the Isomorphism Mapping

When we have a solution, we can use the isomorphism mapping to know which partitions were mapped to which nodes in C, knowing that we can dictate beforehand that the node i in C is going to have a property P_i. This allows us to adapt the results from the algorithm to our application. For example, in Figure 11 after the solution on the left were found we can proceed and use the isomorphism mapping between the orange partition and Node 1 and give the orange partition some properties that were assigned to Node 1 previously like that Node 1 correspond to a desert in the map.

7.3 Changes Needed When Additional Constraints are Added

In the case of political maps, we have no constraints on the solution other than isomorphism, and so the Eigenvalue Method described above is sufficient. However, the heuristic and the goal test need to be modified in other applications that require additional constraints.

The reason we need to do that when we have different requirements/goal is that updating the goal test only can result in a correct solution but the heuristic will not be guiding us toward it. Also, we cannot suffice ourselves with only updating the heuristic. The reason for this is that although the heuristic is steering us toward the properties of the solution we want, the final solution is not guaranteed to have these properties. In particular, if the number of actions from the start to the end is not large enough, the heuristic won't have enough actions through which it can steer us. An example of that is if the initial state turned out to be a solution. So, we must always update the heuristic and the goal test when we add new constraints on the solution.

As of now, our system has the isomorphism goal test and the Eigenvalue heuristic implemented. Next, we will describe some important constraints that we can add to our system, each with a possible implementation of its goal test only. Afterward, in each application, we will mention which constraints that application requires.

7.4 Possible Constraints

7.4.1 Cell-In-Partition Constraint.

Requiring certain nodes in G to end up in certain partitions.

7.4.2 Minimum-Partition-Size Constraint.

Requiring that the size of any partition in the solution is at least some number M.The number M must be set with care so that this partition doesn't starve the other partitions. The goal test could be implemented by checking the size of the minimal size partition after an isomorphism was found.

7.4.3 Maximum-Removed-Nodes-Size Constraint.

Requiring that the number of removed nodes is at most R. This constraint has a similar implementation of the goal test as Minimum-Partition-Size.

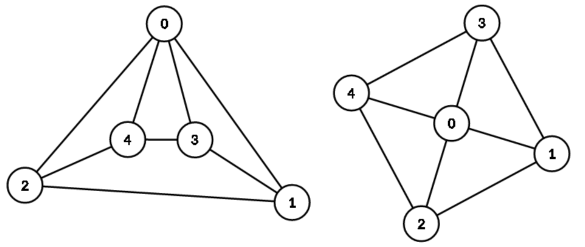

Figure 10: (Left) The graph we had in mind, we sought the nodes 3 and 4 to be the interior and 0,1 and 2 to be the exterior, (Right) The graph embedding we actually got.

7.4.4 Property-Percentage Constraint.
First properties are given to the nodes in C then we require that those nodes in C which have a certain property PROP have at least a percentage of PERC of the nodes in G.To implement the goal test we Use the isomorphism mapping to get those partitions that have the wanted property and we find the percentage of the nodes in these partitions to the total number of nodes in G.

7.4.5 Mapping Constraint.
Requiring that certain partitions in the initial state will end up eventually mapped to certain nodes in C. We can implement the goal test by checking after an isomorphism is found whether the chosen partitions were actually mapped to the required nodes in C. If all the nodes in one partition were added to the Cell-In-Partition constraint then we get this same effect of this constraint. Yet another alternative is to use a Labeled Graph Isomorphism algorithm. A labeled graph is a graph in which nodes and edges have labels. The Labeled Graph Isomorphism algorithm additionally requires the nodes and edges with the same label to be mapped to each other. The Non-Labeled Graph Isomorphism algorithm can be seen as a special case of this algorithm in which all nodes and edges have the same label. Mapping-Constraint can be implemented by given the same label to the partition and the node in C to which it should be mapped.

7.5 One Graph, Many Embeddings

For one graph C, many partitions of G are isomorphic to C. Figures 10 and 11 shed some light on this issue. A possible way to enforce an embedding is through using Mapping or Cell-In-Partition constraints.

7.6 Political Maps

Many strategy games are based on or feature a political map like Risk, Crusader Kings 2, Settlers of Catan, Pandemic, Civilization VI and Total War series to name a few. In some of the games above, the player can't move or capture a region unless it's adjacent to a region he already captured. An important strategic element in these games is the amount of edges that a region has to the neighboring regions as this will affect how easy it is to defend/attack/move-to the region and the number of regions that would become captureable next after taking this region. In other games, there are other considerations that are affected by adjacency of the regions such as spread of infections/city expansion/economical bonuses.

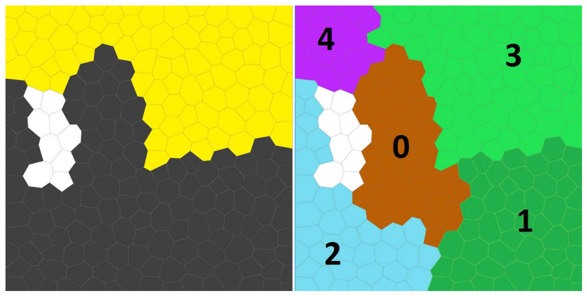

Figure 11: (Right) The solution we had using an isomorphism only goal test. (Left) Same as one on right but using the isomorphism mapping we gave those partitions mapped to EXTERIOR nodes a gray color and those mapped to INTERIOR nodes a yellow color. White nodes are removed nodes

The generation of these political maps were our first motivation and it was the reason we found ourselves discussing graph isomorphism eventually as we observed that the current maps generators lack this constraint of adjacency. Another motivation is that by building a generator that imposes an adjacency/graph constraint we open the path to further work in the future on graph generators that are suitable for strategy games. By allowing the content we want to generate to be controlled through some parameters we can in turn generate these parameters.

7.6.1 Utilizing the Recursive Property.
One feature we found in some of the games above was the recursive nature of the maps, for example, in Risk the map is divided to continents and those continents are then in turn divided to provinces (see Figure 12), in CK2 the map is divided among many factions and these factions in turn control multiple provinces. This motivated us to add this feature to our system and so a constraint graph C_{i+1} is to be provided for each of the nodes in the initial constraint graph C_i, where i is the level of generation, there is a limit though on how deep the nested graph is since eventually the number of nodes in a partition will become too few to satisfy its internal constraint graph.

It is important to note that this recursive property can be used to fight the current inefficiency mentioned in the Results section. For example, when examining the graph of the game Risk (see Figure 12), we find that even though the graph as a whole is complex, the graphs describing the adjacency between the continents is simple and small and so are the graphs of the territories in each of the continents separately.

Remaining is to mention that the algorithm, as it stands, will not necessarily respect the inter-continental edges. For example, looking at the graph in Figure 12, there are multiple edges between nodes from Africa and Europe but they will be replaced by only one abstract edge in the C_i graph which represent that the two continent are connected. That edge in C_i is satisfied if at least two partitions at the $i + 1$ level, each belonging to one of the continents, were adjacent. Because of that, utilizing the recursive structure of the map is only an approximation.

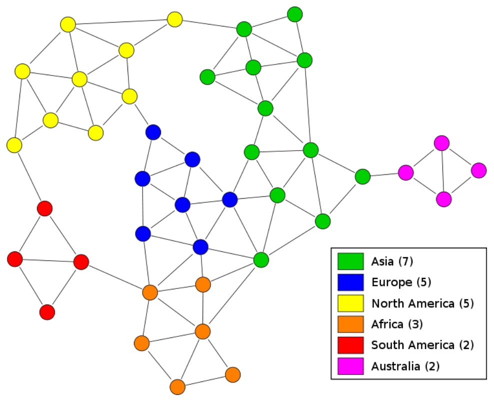

Figure 12: Graph of connections between territories on the Risk game board [5].

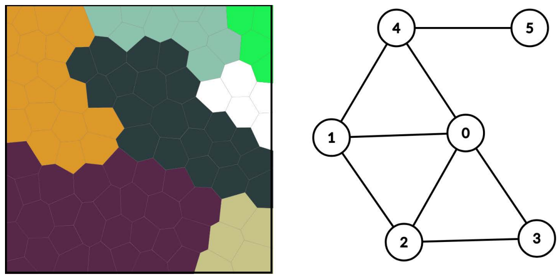

Figure 13: The white nodes are removed nodes, note how removing nodes was one way to satisfy the the edge between nodes 4 and 5 in the graph.

7.6.2 Meaning of Node Removal.
In some cases, as we described earlier, the graph C would be hard or impossible to impose on G without node removal. But this poses the problem of what should we do with the removed nodes as they might mean a gap in, for example, the underlying Voronoi diagram or rectangular grid. Our justification for using node removal was that these nodes could be given a semantic that is based on the domain in which we are applying the algorithm. For example, a removed node might mean a mountain that blocks countries in political maps or walls in a cave, or any meaning that explains its unreachability or unavailability. An illustrating example can be seen in Figure 13. We can use the Cells-In-Partition constraint to ensure that some nodes that are of some economical, religious, etc importance or that are capitals will be in the right partitions.

7.7 Terrain Distribution

We assign terrain properties to the partitions then using the mapping we get through the isomorphism we can assign some partitions as sea, desert, mountain, hills, etc. We might as well use the Property-Percentage constraint to require for example that 60% of the map is made of water. Using isomorphism in terrain distribution allow us to express requirements of placing deserts for example beside villages but not beside grass lands.

7.8 Further Control

Here we present two ideas to enable the users to further control the algorithm in addition to the constraints mentioned earlier. We will illustrate them by solving the problem of enforcing that an ocean should exist between some of the continents in Figure 12. The first idea is to apply the changes on the constraint graph. For example, in our case we include a node in the continents graph to represent an ocean and the nodes of all the continents which are linked to this ocean will have an edge with it. After the partitioning is found, the isomorphism mapping from Section 7.2 can be used to find the partition mapped to this ocean node. The second approach is a post-generation process in which we find the nodes in G that lies on the borders between the continents between which an ocean must exist. The general idea is to carve from the partitions the nodes we want. We also used this idea in the example in Figure 16. The main disadvantage of the first approach is that it increases the search space size but its advantage over the second is that we guarantee that enough nodes were assigned for our purpose whereas we might resort to making the graph G larger in the second approach to be able to safely reserve portions of it.

7.9 Converting a Mission Graph to a Game Space

As we mention in the Section 2, generating game levels can be accomplished by converting a Mission Graph describing what the player is going to do to a game space where that will take place. In the work by Dormans [4], the missions are generated using graph grammars which operate on graph nodes and edges instead of strings and the game space is then generated using Shape Grammers. Furthermore, in the work by Van der Linden et al. [22], levels for the game Dwarf Quest[8] are generated by first embedding the Mission Graph into a 2D grid and then placing a room in every cell where a graph node was placed. Another work which handles the same general problem of converting an abstract graph into a game space is [14] who uses a Force-Based graph drawing on the graph. For each graph node, points are created to represent a room (points arranged as a hexagonal, rectangle or an irregular polygon) as in Figure 14 and for every edge, points are created to represent corridors. After that, points are also created for the empty spaces. Finally, a Voronoi diagram is generated using all these points.

What we are aiming to do is to partition a graph such that the adjacency between the partitions is described by the constraint graph. If we regard the Mission Graph mentioned previously as our constraint graph, then our work shares a similar aim as these approaches above.

The last approach is the closest to ours among the approaches described above, but also it has an interesting relation to our algorithm. Where as we start with generating the Voronoi diagram and then impose the constraint graph on its graph, the constraint graph, in this approach, is first drawn and then the Voronoi diagram is generated. This also shows a way in which graph drawing and graph partitioning with isomorphism constraint are linked. Also, in the last approach, the nodes in the constraint graph are mapped to rooms (which can vary in size and shape based on the points

[8](Wild Card Games) www.dwarfquestgame.com/

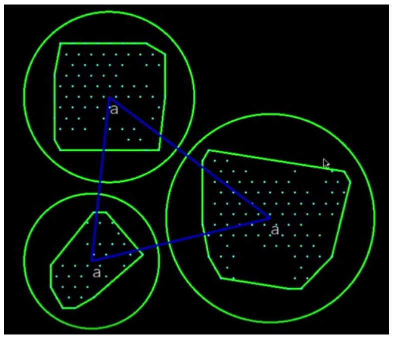

Figure 14: Reproduced with the owner (naughtty) permission[9].

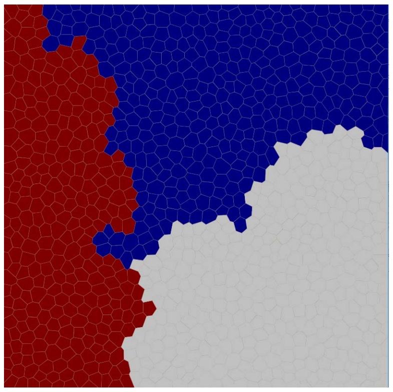

Figure 15: The result of our algorithm are whole regions.

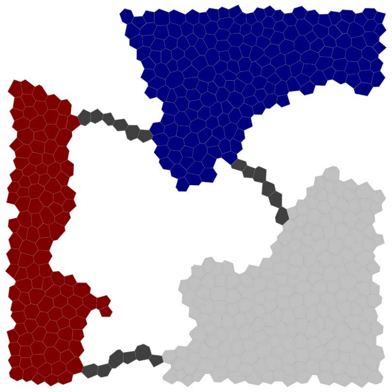

Figure 16: We can shape these regions. e.g. here, we shrank each region from all directions and then created paths between them. This example could be seen as a start for dungeon generation.

creation step) while in ours the nodes are mapped to whole regions/partitions of the basic graph. Each region can then be carved by node removal to the shape we want. (See Figure 15 and Figure 16) Alternatively, we can use the regions we found as borders inside which we can place other types of content. We can then link contents in the regions between which an edge exists in the constraint graph. To obtain these borders, we can merge the Voronoi cells belonging to the same partition or treat the dual graph of the Voronoi diagram as a polygon and pick the edges that are on the borders of the region. It must be noted, though, that the obtained polygons in the two cases might be concave.

7.10 Linking a Mission Graph to a Game Space

We don't need to create the game space in the process, due to the general nature of our algorithm, we can divide an already created game space using the Mission Graph given that the game space has a graph representation. We assume that the relation between the Mission Graph and the game space is one-to-many, i.e. one task is linked to multiple nodes in the game space graph.

Using our algorithm two problems might appear, the first is that there is no guarantee that the node corresponding to the starting task will actually be mapped to the region in the game space where the player is ought to start (Same applies to the last task and the final region of the game space). To enforce that, a Mapping-Constraint might be used.

Secondly, the transition between each task in the Mission Graph is sometimes realized through an actual boundary in the game space like a gate, a portal or a corridor. The partitions we acquire has no regard to these boundaries. This is true if the graph representation of the game space is fine-grained (e.g. raw rectangular or Voronoi cells) as opposed to a representation where every logical region in the game context (e.g. a room) is a one node. A more coarse representation might be acquired first to evade this problem. Alternatively, we may add a constraint that all the entry/exit points of a region must end up in the same partition (but then, as we mentioned in Section 7.3, the heuristic and the goal test must be updated). We can, as well, satisfy this constraint as a post-partitioning step where we assign every logical region to the partition that has, for example, the largest portion of its nodes.

Finally, until now we have been using a deterministic algorithm to obtain the initial state and different graphs G were used to introduce variation. If the same graph G can't be changed (as might be the case for the game space graph) we can instead use a randomized algorithm to obtain the initial state. This can be an algorithm akin to K-Means clustering where $|C|$ nodes are chosen at random to be the seeds of the partitions. The remaining nodes are then added to the partition whose seed is the closest. This algorithm will produce partitions that have more irregular shapes than the one we have been using. Furthermore, note that new random seeds will be used in every run when the Restart Policy (Discussed in Section 5.1) is used.

[9]A snapshot from www.youtube.com/watch?v=RAtdFKiqs34&feature=youtu.be

8 SUMMARY AND FUTURE WORK

To summarize, given a planar graph G we presented an algorithm to partition it such that the quotient graph is isomorphic to a constraint graph C. We used the A-Star search algorithm with a heuristic that is based on the isospectrality of isomorphic graphs. To handle the case when the basic graph G is large (which increases the size of the search space), we introduced a coarsening step which will partition G into a new graph G' that is smaller in size. We also discussed why choosing an inappropriate new size might over-limit the search space (when new size is too small) or require more computations than necessary (when new size is too big). After that we showed statistics that suggest the sensitivity of the algorithm to the initial state/partitioning. To handle this, we employed a Restart Policy which reduced the algorithm unpredictability.

To handle the inefficiency when a constraint graph is hard to impose, we suggested that any recursive property or acceptable approximations in the desired application can be utilized. Additionally, we illustrated how a mixed-initiative tool can shift the focus of its users to the end result regardless of what computations-saving means they used to acquire it. When deployed as an Online PCG, extra care must be taken to allow only the constraint graphs and the initial states which we know to be solvable efficiently. Furthermore, we suggested more constraints that can tune the output to our needs and justified why both the heuristic and the goal test must be updated when any additional constraints such as these are added. Finally, we showed multiple applications to the algorithm in PCG which included: the generation of political maps, terrain distribution and converting/linking a Mission Graph to a game space. We also showed how can the additional constraints we presented aid in achieving them.

In the future, we wish to work on making the algorithm more efficient. Furthermore, as clear from the Section 7.4, implementing the heuristics for the additional constraints is going to allow us to test and expand the possible applications we speculated above. We also aim to gain a more solid understanding of what makes a constraint graph hard to impose. This, we believe, will allow us to write a generator that generates random constraint graphs that are efficient to impose. We wish to explore other restart policies than the one we used. Finally, we want to explore other heuristics that might perform better than the current one.

ACKNOWLEDGMENTS

I would like to thank Dr. Wasfi G. Al-Khatib for his advice, mentorship and for lending me his ears, Mr. Anas Al-Baghajati and Mr. Faisal Alvi for their support and the reviewers of this paper for their insightful comments.

REFERENCES

[1] 2016. JGraphT. http://www.jgrapht.org/. (2016).

[2] Azgaar. 2017. Fantasy Map Generator. (2017). https://azgaar.wordpress.com/

[3] Tom Betts. 2014. *Procedural Content Generation.* John Wiley & Sons, Inc., 62–91. https://doi.org/10.1002/9781118796443.ch2

[4] Joris Dormans. 2010. Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games (PCGames '10).* ACM, New York, NY, USA, 1:1—-1:8. https://doi.org/10.1145/1814256.1814257

[5] Fanblade. 2009. Risk Game Graph. (2009). https://commons.wikimedia.org/wiki/File

[6] Carla Gomes, Bart Selman, and Nuno Crato. 1997. Heavy-tailed distributions in combinatorial search. *Principles and Practice of Constraint Programming-CP97* (1997), 121–135.

[7] Peter E Hart and J Nils. 1968. Formal Basis for the Heuristic Determination eijj ,. 2 (1968), 100–107.

[8] Bruce Hendrickson and Robert Leland. 1995. A Multilevel Algorithm for Partitioning Graphs. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (Supercomputing '95).* ACM, New York, NY, USA. https://doi.org/10.1145/224170.224228

[9] Peter Webb Ronald F. Boisvert Bruce Miller Roldan Pozo Joe Hicklin, Cleve Moler and Karin Remington. 2012. Java Matrix Package. https://github.com/carlsonp/JAMA. (2012).

[10] George Karypis and Vipin Kumar. 1998. Multilevel Algorithms for Multi-constraint Graph Partitioning. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (SC '98).* IEEE Computer Society, Washington, DC, USA, 1–13. http://dl.acm.org/citation.cfm?id=509058.509086

[11] Danai Koutra, Ankur Parikh, Aaditya Ramdas, and Jing Xiang. 2011. Algorithms for Graph Similarity and Subgraph Matching. (2011).

[12] Antonios Liapis, Gillian Smith, and Noor Shaker. 2016. *Mixed-initiative content creation.* Springer International Publishing, Cham, 195–214. https://doi.org/10.1007/978-3-319-42716-4_11

[13] Orlando Moreira, Merten Popp, and Christian Schulz. 2017. Graph Partitioning with Acyclicity Constraints. *CoRR* abs/1704.00705 (2017).

[14] Naughty. 2013. Graph Grammar and Voronoi based Level Generation for a Roguelike. (2013). www.reddit.com/r/gamedev/comments/1bne5o/graph

[15] Martin O'Leary. 2015. Generating fantasy maps. (2015). www.mewo2.com/notes/terrain/

[16] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 26, 10 (oct 2004), 1367–1372. https://doi.org/10.1109/TPAMI.2004.75

[17] Amit Patel. 2010. Polygonal Map Generation for Games. (2010). www-cs-students.stanford.edu/

[18] Rüdiger Lunde, Ciaran O'Reilly, janphkre , Mayank Kumar and Andrey Mochalov. 2015. AIMA3e-Java. https://github.com/aimacode/aima-java. (2015).

[19] Julian Togelius, Georgios N Yannakakis, Kenneth O Stanley, and Cameron Browne. 2010. Search-based Procedural Content Generation. In *Proceedings of the 2010 International Conference on Applications of Evolutionary Computation - Volume Part I (EvoApplicatons'10).* Springer-Verlag, Berlin, Heidelberg, 141–150. https://doi.org/10.1007/978-3-642-12239-2_15

[20] Jacobo Torï£¡n and Fabian Wagner. 1997. The Complexity of Planar Graph Isomorphism. (1997).

[21] trickl. 2016. Trickl-Graph. https://github.com/trickl/trickl-graph. (2016).

[22] Roland Van der Linden, Ricardo Lopes, and Rafael Bidarra. 2013. Designing procedurally generated levels. In *Proceedings of the the second workshop on Artificial Intelligence in the Game Design Process.*

[23] Rafael Van Driessche and Dirk Roose. 1995. Dynamic load balancing with a spectral bisection algorithm for the constrained graph partitioning problem. In *High-Performance Computing and Networking: International Conference and Exhibition Milan, Italy, May 3–5, 1995 Proceedings*, Bob Hertzberger and Giuseppe Serazzi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 392–397. https://doi.org/10.1007/BFb0046658

[24] Luh Yen, Francois Fouss, Christine Decaestecker, Pascal Francq, and Marco Saerens. 2007. *Graph Nodes Clustering Based on the Commute-Time Kernel.* Springer Berlin Heidelberg, Berlin, Heidelberg, 1037–1045. https://doi.org/10.1007/978-3-540-71701-0_117

[25] Zhiping Zeng, Anthony K. H. Tung, Jianyong Wang, Jianhua Feng, and Lizhu Zhou. 2009. Comparing stars: On Approximating Graph Edit Distance. *Proceedings of the VLDB Endowment* 2, 1 (2009), 25–36. https://doi.org/10.14778/1687627.1687631