

# Calculating the minimum planar graph

Sam Doctolero and Alex M Chubaty

2026-07-06

## Contents

Overview . . . . .	1
Finding the Minimum Planar Graph . . . . .	2
Technical reference to the MPG engine written in C++ . . . . .	2
Terminology . . . . .	3
Data Structures . . . . .	3
Type Definitions . . . . .	5
The <code>Engine</code> Class . . . . .	5
How to Use the Engine . . . . .	13
Debugging the package C++ code . . . . .	15
References . . . . .	16

## Overview

The Minimum Planar Graph (MPG) is a spatial representation of a mathematical graph or network that is useful for modelling dense two-dimensional landscape networks (see Fall et al. 2007). It can efficiently approximate pairwise connections between graph nodes, and this can assist in the visualization and analysis of how a set of patches is connected. The MPG also has the useful property that the proximity, size and shape of patches in the network combined with the pattern of resistance presented by the landscape collectively influence the paths among patches and the end-points of those links. In this sense the MPG can be said to be spatially-explicit, and therefore to be a property of the entire landscape under analysis (or alternatively a property of the digital resistance map used to represent the landscape).

The MPG achieves this spatially-explicit property by finding Voronoi polygons that describe regions of proximity in resistance units around focal patches (Fall et al. 2007). The algorithm that is used to find the Voronoi boundaries and approximate the least cost-paths between patches and their end-points is described below.

## Finding the Minimum Planar Graph

In `grainscape`, the boundaries of Voronoi polygons are found by using a spreading or marching algorithm. This is done beginning in each perimeter cell of a patch and spreading out to adjacent cells that are not part of any patch and have not been visited yet by the algorithm. These cells are then given a patch ID to mark the Voronoi territory. A Voronoi boundary is found when a cell is visited twice by two different Voronoi territories or IDs originating from different patches (see Fall et al. 2007).

Using a marching algorithm to find the Voronoi boundaries makes it possible to implement a linking algorithm that can run in parallel with the marching algorithm. As a cell is spread into (let's call it a child cell) it then creates a link or connection between the child cell and the cell that it spread from, which we call a parent cell.

Finding the least-cost path in this way is only possible because the algorithm stores the child cells (which will eventually become parent cells) in a queuing table that sorts the cells in a certain order. The child cells are sorted by increasing effective distance (i.e., resistance or cost) between the child cell and their origin cell, the perimeter cell that the connection originally spawned from. A link or path between patches is then created at the first Voronoi boundary between two patches.

Two refinements keep these least-cost paths and the tessellation accurate. First, every patch (source) cell begins spreading at the same minimum cost, regardless of the resistance of the cell it happens to overlap; otherwise a patch sitting on a high-resistance cell would spread late and lose territory, biasing the Voronoi tessellation. Second, a patch-to-patch link is only recorded once the boundary cell has *settled* — that is, once it has itself spread and its stored connection back to its origin patch is as cheap as it will get — so that an early, expensive boundary cannot lock in a sub-optimal link before a cheaper route has had a chance to form. When two patches meet, the stored connections are followed back to each patch to assemble the candidate path; if a link between that pair of patches already exists, the cheaper of the two is kept, and a link that merely retraces a cheaper two-step route through a third patch is discarded as redundant.

The MPG algorithm has the following general steps. These are represented in more detail in a flow chart in Figure 1.

1. Create Active Cells.
2. Check if the Active Cells are ready to spread.
3. Spread to all 4 adjacent cells for all the Active Cells that ready to spread.
4. The cells that have been recently spread in to become new Active Cells.
5. Repeat.

The linking algorithm is embedded within the spreading functions of the MPG algorithm. When an `ActiveCell` spreads, a link map creates a connection between the parent `ActiveCell` to the new (child) `ActiveCell`. Linking is assisted by the queue when finding the least-cost paths.

## Technical reference to the MPG engine written in C++

The following is intended to provide an overview of the C++ engine provided by the package that implements the MPG algorithm. It may be useful for those who wish to implement MPG extraction in other programming languages. Reading and interpretation of this section is not required for the use of `grainscape` in R. An interface to this code has been abstracted to R functions using the `Rcpp` package (Eddelbuettel et al. 2024).

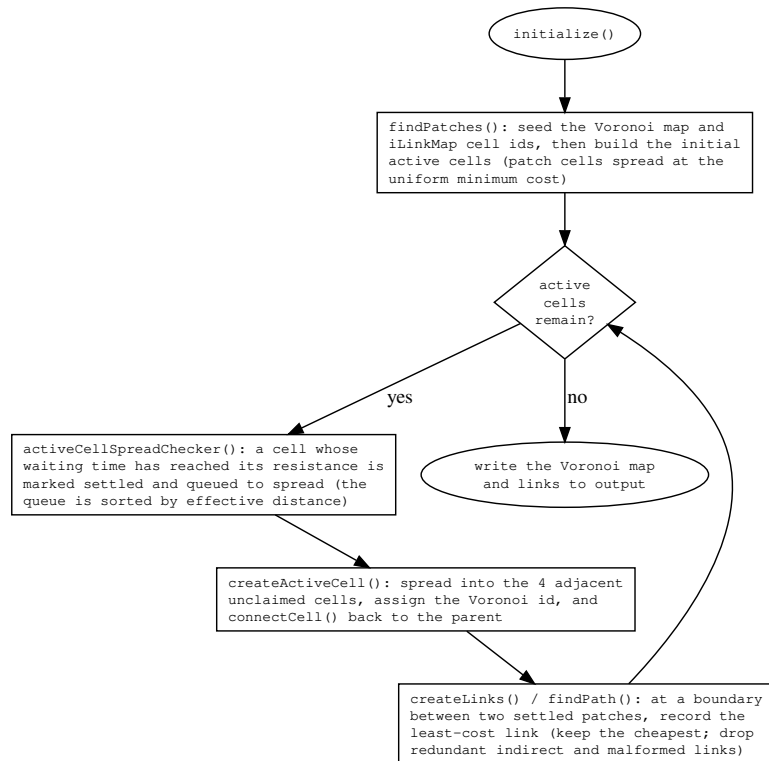


Figure 1: Overview of the MPG algorithm.

## Terminology

- *Cell*: A box or element in a map.
- *Active Cell*: A type of cell that is currently being evaluated. It refers to the child cell mentioned above.
- *Time*: An index of the iteration.
- *Object*: An instance of a certain data type, class, or data structure (*e.g.*, `Cell c`, refers to an object `c` of type `Cell`).

## Data Structures

- `Cell`: stores its own position (row and column) and an ID.
- `ActiveCell`: inherits the properties of a `Cell` and has its own properties: `time` (an iteration index), `distance` (the Euclidean distance back to its origin cell), `resistance`, `parentResistance` (the resistance of the cell it spread from), and `originCell`. This type of cell is used to keep track of which cells are currently being evaluated.
- `LinkCell`: inherits the properties of a `Cell` and has its own properties such as `cost`, `distance`, `fromCell`, and `originCell`. This type of cell is used to create `LinkMap`.
- `ActiveCellHolder`: a type of container that stores a vector of `ActiveCells` in an order.
- `ActiveCellQueue`: contains an `ActiveCellHolder`. Its main purpose is to properly store the `ActiveCellHolder` in a vector in order of increasing effective distance (*i.e.*, resistance or cost).
- `InputData`: contains all the data that is needed for the engine to operate. The user of the

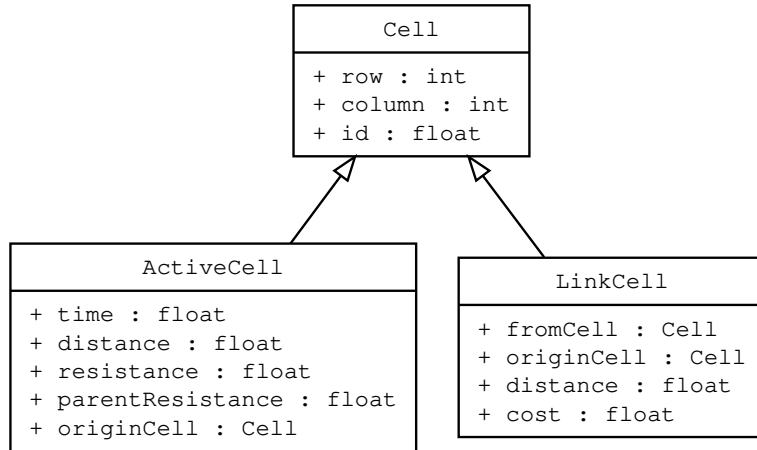


Figure 2: Schematic representation of Cell type data structures. An open triangle denotes inheritance.

engine has to create an instance of it and initialize all the properties before giving the address of the object to the engine’s constructor.

- **Link:** stores all the links (directly and indirectly) between the patches. Links are given a negative ID to distinguish them from patch IDs.
- **OutputData:** similar to **InputData** but it acts as a container for all the data that are calculated by the engine and gives that data to the user.
- **Patch:** a patch or a cluster are the habitats that are found in the resistance map, given a value for habitat.

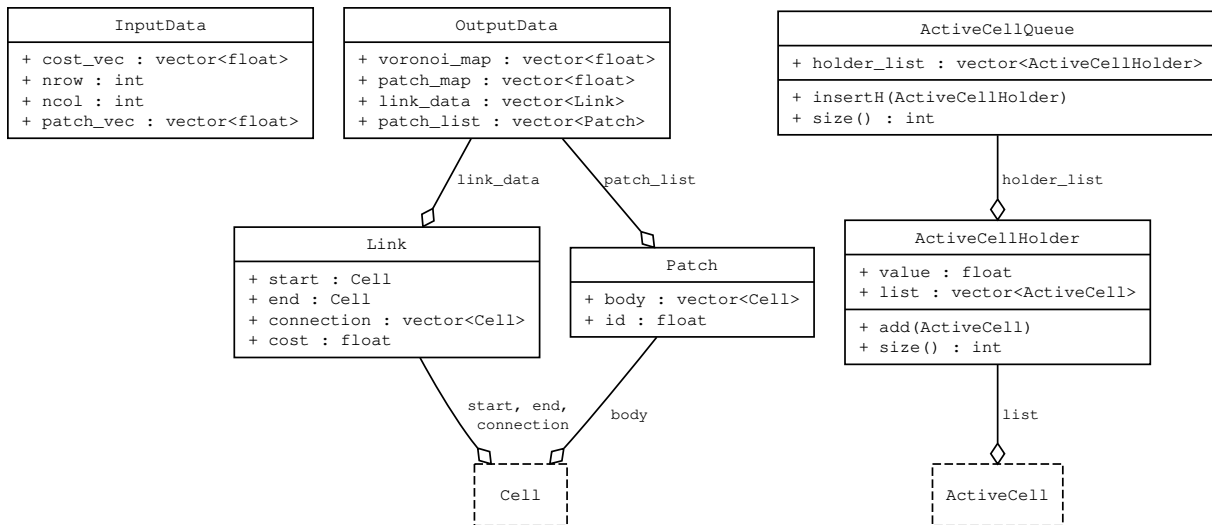


Figure 3: Schematic representation of additional data structures. An open diamond denotes composition (a ‘has-a’ relationship); dashed boxes are the Cell types shown above.

## Type Definitions

- `lcCol`: a vector of `LinkCells`.
- `LinkMap`: a vector of `lcCols`, which in turn creates a `Map`. This type stores the connections between cells.
- `flCol`: a vector of floating point values.
- `flMap`: a vector of `flCol`, which in turn creates a `Map` that contains floating point values in each element or cell.
- `boolCol`: a vector of boolean values.
- `boolMap`: a vector of `boolCol`, which in turn creates a `Map` of booleans. This type stores the `settled_map` (see below).

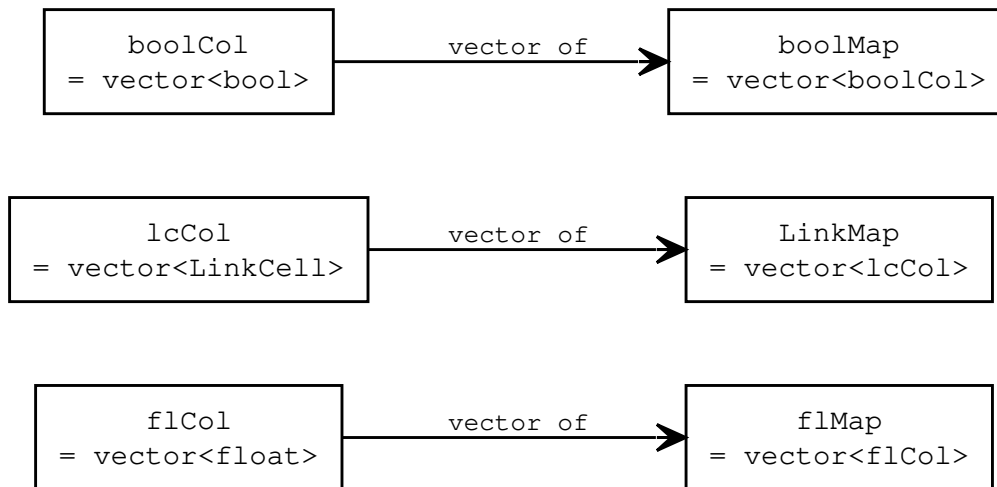


Figure 4: Schematic representation of type definitions.

## The Engine Class

The main calculator of the program. It creates the minimum planar graph (MPG) using the MPG algorithm, finds least cost links or paths, and finds patches or clusters.

## Fields/Properties

Property	Data Type	Description
<code>in_data</code>	<code>InputData</code> Pointer	Points to an <code>InputData</code> object. This is where the engine gets all the initialization values from.
<code>out_data</code>	<code>OutputData</code> Pointer	Points to an <code>OutputData</code> object. The engine stores all the calculated values in this variable.
<code>maxCost</code>	Float	The maximum resistance or cost in the resistance map.
<code>costRes</code>	Float	The minimum resistance or cost in the resistance map.

Property	Data Type	Description
<code>active_cell_holder</code>	<code>ActiveCellQueue</code>	Holds or stores all the <code>ActiveCells</code> .
<code>temporary_active_cell_holder</code>	<code>ActiveCellQueue</code>	Similar to <code>active_cell_holder</code> , except it acts as an intermediate or temporary holder of <code>ActiveCells</code> . Required for vector resizing and comparing.
<code>spread_list</code>	vector of <code>ActiveCells</code>	Stores all the <code>ActiveCells</code> that are ready to spread to all 4 adjacent cells, if possible.
<code>iLinkMap</code>	<code>LinkMap</code>	A map that keeps track of all the connections between cells due to the spreading and queuing functions.
<code>voronoi_map</code>	<code>f1Map</code>	A map that contains floating point values, it stores the Voronoi boundaries/polygons.
<code>cost_map</code>	<code>f1Map</code>	A map that contains the resistance or cost in each cell/element.
<code>settled_map</code>	<code>boolMap</code>	Tracks which cells have been fully processed (have appeared in <code>spread_list</code> and spread to their neighbours). <code>createLinks</code> only records a patch-to-patch link when the boundary cell is settled, so the cheapest path has a chance to form first.
<code>initialized</code>	<code>Bool</code>	Indicates whether the engine has been successfully initialized and is ready to run via <code>start()</code> .
<code>error_message</code>	<code>Char Pointer</code>	Stores the error messages that occur in the engine. It acts as a way to diagnose problems in the engine.

**Methods/Functions** After instantiating an `Engine` object, the connectivity engine is run by first calling `initialize()` and then `start()`. The call graphs for each of these principal functions are presented below.

Engine
<pre> - in_data : InputData* - out_data : OutputData* - maxCost : float - costRes : float - active_cell_holder : ActiveCellQueue - temporary_active_cell_holder : ActiveCellQueue - spread_list : vector&lt;ActiveCell&gt; - iLinkMap : LinkMap - voronoi_map : flMap - cost_map : flMap - settled_map : boolMap - initialized : bool - error_message : char* - error_message_size : int </pre>
<pre> + Engine() / ~Engine() + initialize() : bool + start() : void + emax() / emin() / calcDistance() [static] - findPatches() / getIndexFromList() / combinePatches() - activeCellSpreadChecker() / createActiveCell() - createLinks() / findPath() / parseMap() / lookForIndirectPath() - connectCell() / cellIsZero() / cellsEqual() / outOfBounds() - updateOutputMap() / writeErrorMessage() </pre>

Figure 5: Schematic representation of the Engine class. A leading + marks a public member and - a private member; related methods are grouped on a line. Full signatures are given in the tables below.

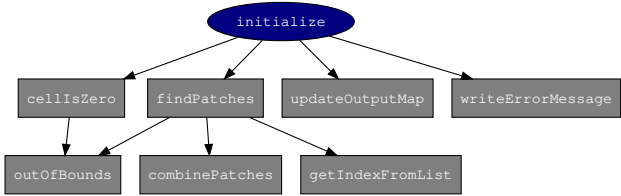


Figure 6: Call diagram for Engine::initialize()

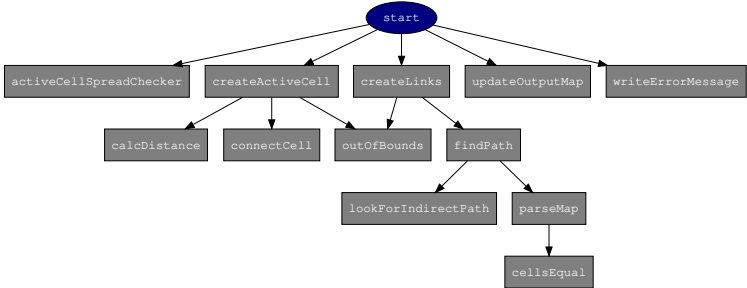


Figure 7: Call diagram for Engine::start()

**Public Functions** These are the functions that are visible to the user.

---

<b>Function</b>	<b>Return Type</b>	<b>Input Arguments</b>	<b>Description</b>
<code>Engine</code>	Instance of an <code>Engine</code>	None	Default <code>Engine</code> constructor.
<code>Engine</code>	Instance of an <code>Engine</code>	<code>InputData</code> Pointer, <code>OutputData</code> Pointer, <code>Char</code> Pointer	<code>Engine</code> constructor.
<code>initialize</code>	Boolean	None	Prepares the engine for calculation.
<code>start</code>	Void	None	Runs the MPG algorithm.

---

**Patch Finding Functions** The functions are responsible for identifying the patches (clusters) in a resistance map, given a value for a habitat.

<b>Function</b>	<b>Return Type</b>	<b>Input Arguments</b>	<b>Description</b>
<code>findPatches</code>	Void	Nothing	Finds all the patches in the patch vector and assign patch IDs.
<code>getIndexFromList</code>	Int	Float, Vector of Patches	Finds the index in the vector of patches that the given ID correspond to.
<code>combinePatches</code>	Int	Int, Int, Vector of Patches	Given two indices and the list of patches. Extract the two patches from the list and combine those two into one patch. Insert the new patch into the list and return the index value of the new patch.

**Linking Functions** These functions create the links between cells and finds the least cost (direct or indirect) paths between patches.

Function	Return Type	Input Arguments	Description
<code>createLinks</code>	Void	ActiveCell Pointer, Int, Int	At a Voronoi boundary, attempts a patch-to-patch link, but only when the neighbouring boundary cell is <i>settled</i> so the cheapest path can form first. Calls <code>findPath</code> .
<code>findPath</code>	Void	LinkCell, LinkCell, Vector of Links	Finds the least-cost path between two patches: assembles the candidate path and its cost, rejects malformed links (an unset <code>-99</code> endpoint), and for an existing patch pair keeps only the cheaper link.
<code>connectCell</code>	Void	ActiveCell Pointer, Integer, Integer, Float	Connects the child cell to the parent cell in <code>iLinkMap</code> .
<code>parseMap</code>	Cell	LinkCell, Link	Follows the connections from a starting <code>Cell</code> until it reaches a patch, accumulating the path cost; the last (patch) cell is returned.
<code>lookForIndirectPath</code>	Bool	Vector of Links, Link	Searches all pairs of existing links for a cheaper two-hop route (A to X to B), ignoring non-patch ( <code>-99</code> ) pivots; returns <code>false</code> and updates the path when a cheaper indirect route is found.

**Common Functions** Common functions are used in almost all of the functions in the engine.

---

<b>Function</b>	<b>Return Type</b>	<b>Input Arguments</b>	<b>Description</b>
<code>outOfBounds</code>	Bool	Int, Int, Int, Int	Checks to see if the given row and column is still within the resistance map's dimensions.
<code>cellsEqual</code>	Bool	Cell, Cell	Compares the two Cells' row and column if they match.

---

**Static Functions** Static functions are functions that can be used without declaring an object of the class.

<b>Function</b>	<b>Return Type</b>	<b>Input Arguments</b>	<b>Description</b>
<code>emax</code>	Float	Vector of Floats	Finds the maximum value from the vector of floating point values
<code>emin</code>	Float	Vector of Floats	Finds the minimum value from the vector of floating point values
<code>calcDistance</code>	Float	Cell, Cell	Finds the Euclidean distance between two Cells

## How to Use the Engine

1. Create `InputData` and `OutputData` objects.
2. Initialize the `InputData` object's fields. Keep in mind that the vectors in the `InputData` and `OutputData` structures are all of type `float`.
3. Create an array of `Char` with the length of `MAX_CHAR_SIZE` or a larger value.
4. Create an `Engine` object and give the address of the `InputData` and `OutputData` objects, the `Char` array and the size of the array as arguments.
5. Call the initialization function from the `Engine` object.
6. If the initialization is successful, call the start function from the `Engine` object. If the initialization is not successful, the array of char will contain the reason for the initialization failure.
7. Once the engine is done calculating, extract all the fields needed in the `OutputData` object.

A snippet of C++ code is shown on the next page as an example.

Note that the current `Engine` has two lines of code that are meant for interfacing with R via `Rcpp` (Eddelbuettel et al. 2024). In order to make the `Engine` run with any programming or scripting language, remove those two lines. One of them is an `include` statement for `Rcpp`, at the very top of source code, and the other is inside the `start` function, the first line inside the `while` loop. Those two lines are convenient for R users when they want to interrupt or stop the MPG algorithm safely, without crashing their console and possibly losing their data.

```

vector<float> EngineInterface(vector<float> resistance, vector<float> patches,
                             int nrow, int ncol)
{
    // InputData and OutputData objects
    InputData inObj;
    OutputData outObj;

    // Initialize InputData object
    inObj.cost_vec = resistance;
    inObj.nrow = nrow;
    inObj.ncol = ncol;
    inObj.patch_vec = patches;

    // Array of chars with a size of MAX_CHAR_SIZE
    char error[MAX_CHAR_SIZE];

    // Engine object while passing in the InputData and OutputData objects'
    // address and the array of chars
    Engine engineObj(&inObj, &outObj, error, MAX_CHAR_SIZE);

    // Initialize the engineObj;
    // If it fails output the reason why and exit the function
    if (engineObj.initialize() == false)
    {
        cout << error << endl;
        return outObj.voronoi_map;
    }

    // start the calculation
    engineObj.start();

    //extract the data needed, in this case the voronoi_map
    return outObj.voronoi_map;
}

```

## Debugging the package C++ code

Since we are using clang++ to compile the code locally, we need lldb installed.

```
cd ~/GitHub/grainscape

## start R in debug mode
R -d lldb

## from the lldb shell, set breakpoint(s) in functions for debugging
# breakpoint set --name initialize
# breakpoint set --name start
# breakpoint set --name activeCellSpreadChecker
breakpoint set --name createActiveCell
breakpoint set --name createLinks
breakpoint set --name findPath
breakpoint set --name lookForIndirectPath

## start R
run

## from the R shell, run the code needed to debug
devtools::dev_mode()
devtools::load_all()

## RUN ADDITIONAL R CODE FOR DEBUGGING
```

Some useful lldb shell commands for stepping through code:

```
## run next line
n

## continue
c

## skip
s

## step up into the parent frame
up

## inspect variables the current frame
frame variable

## inspect a particular value or evaluate code
expr i
expr spread_list[23];
```

See <https://lldb.llvm.org/use/tutorial.html#controlling-your-program> for more help.

## References

- Eddelbuettel, Dirk, Romain Francois, JJ Allaire, et al. 2024. *Rcpp: Seamless r and c++ Integration*. <https://www.rcpp.org>.
- Fall, Andrew, Marie-Josée Fortin, Micheline Manseau, and Dan O'Brien. 2007. "Spatial Graphs: Principles and Applications for Habitat Connectivity." *Ecosystems* 10 (3): 448–61. <https://doi.org/10.1007/s10021-007-9038-7>.