Ivy for Grasshopper Manual

version 0.860

This is a tentative manual for Ivy. It contains the description of every tool that comes with the package along with the description of the relevant inputs and outputs. Also included are a few usage tips for the crucial components.

The tools in Ivy are grouped into nine sections and are laid out on the Grasshopper ribbon in a similar fashion to their expected use. In other words, the sequence you might employ them might resemble the enumeration of sections on the ribbon.

Creating a MeshGraph – Adding weight to the MeshGraph – Primary Segmentation (Tree Making) – Secondary Segmentation – Iterative Segmentation – Special Segmentation – Fabrication.

Additional helpful tools are grouped in the last two sections Mesh Info and Other Tools.

1. MeshGraph creation.

The research behind Ivy is based on graph theory and as such the main concept and the data backbone of the plug-in is an object that encodes the properties of a graph (nodes and edges) constructed on top of an input mesh. The first section in Ivy contains the tools for creating graphs from meshes (called from now MeshGraphs or just simple graphs) and extracting basic information from those graphs.

A **MeshGraph** is a set of collections of data directly derived from the mesh geometry/topology and enhanced by exterior data associated with the mesh by the user. The most important collections inside the MeshGraph data object are the list of nodes and the list of edges connecting those nodes. In Ivy the MeshGraph nodes are derived from the mesh faces and the MeshGraph edges are constructed initially from the non-naked topology edges of the mesh. This makes the initial MeshGraph constructed from a mesh, the dual graph of the respective mesh. However, the MeshGraph is not bound as the dual of the mesh forever. The nodes and edges collections can lose or gain members according to the rules enforced by the other tools in Ivy.

The MeshGraph **nodes** are data objects too and they encode more than their geometric mesh counterpart. A typical MeshGraph node contains references to connecting edges, neighboring nodes, a weight value and many other bits of data relevant to different Ivy operations.

In a similar fashion, **edges** are also complex data objects maintaining references to connecting nodes, a weight value, angle, bend type, as well as a link to their geometric mesh counterpart - the topological edge.

Graph from mesh (GraphMsh) Creates a MeshGraph from a mesh specified by the user

Inputs:

(I) Input Mesh– the mesh used as a basis for the MeshGraph

Outputs:

(G) MGraph– the MeshGraph object

Graph to mesh (Graph2Msh) Extracts the mesh from a MeshGraph object. Only the nodes present in the MeshGraph will be translated back into mesh faces.

Inputs:

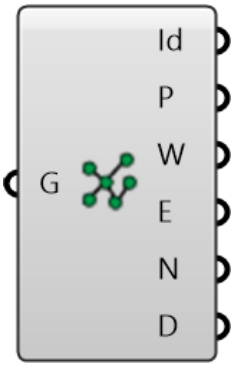
(G) MeshGraph– the MeshGraph object

Outputs:

(M) Mesh - The mesh constructed from the nodes of the MeshGraph

---

MeshGraph (MGraph) This is the container for temporary storage of the MeshGraph data type. Because the MeshGraph data object has inbuilt automatic data casting it can be used to convert from and to the standard mesh type. Just plug in a mesh as an input or output. This works for every MeshGraph input or output and for all Ivy tools.



Graph Nodes (Nodes) This tools extracts the node information for each node present in the input graph. The outputs are populated if the respective information is present in the node

Inputs:

(G) MeshGraph – the MeshGraph object

Outputs:

(Id) Mesh Id – the number of the face in the face collection of the original mesh

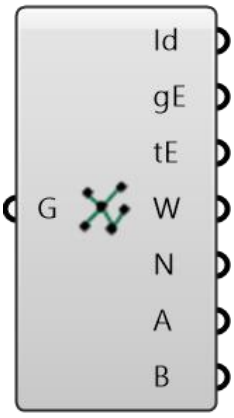
(P) Point – a point representing the geometric location of the node. This is the usually the center of the face polygon

(W) Weight – a floating point precision number between 0 and 1 expressing the weight of the node in the graph.

(E) Edges – a list of graph edges (their mesh Id) connecting to this node

(N) Neighbors – a list of nodes (their mesh Id) directly connected by edges to this node inside the MeshGraph.

(D) Depth – the depth of the node inside a tree graph (will be 0 if graph is not a tree)



Graph Edges (Edges) This extracts the edge information for edges present in the input graph. The outputs are populated if the respective information is present in the edge.

Inputs:

(G) MeshGraph – the MeshGraph object

Outputs:

(Id) Mesh Id – the number of the edge in the topological edge collection of the original mesh

(gE) Graph Edge – the polyline representing the geometric connection between two graph nodes. The node points are extracted by the previous component

(tE) Topo Edge – the geometric representation of the mesh edge this MeshGraph edge references. This is a line.

(W) Weight – a floating point precision number between 0 and 1 expressing the weight of the edge in the graph.

(N) Nodes – the graph nodes this edge connects. The nodes are expressed through their unique mesh Id.

(A) Angle – the dihedral angle in radians between the faces of the mesh connected by the corresponding topological edge. Values range from 0 to Pi.

(B) Bend – the type of bend considering the normals of the mesh. 1=Flat; 2=Convex; 3=Concave.

2. Adding weight.

This section contains a set of tools used to add additional weight data to the mesh graph collections (edges and nodes). Most of the tools work for edges but there are some tools for specifying node weight. Weight is a useful concept for differentiating between the edges or nodes of the mesh. The weight data is used by the segmentation algorithms to decide where to perform edge removals either to partition the mesh or convert the graph into a tree. The weight data can come from any numerically expressible source within the Grasshopper environment. This includes mesh information (like dihedral angle, or face size) or outside information (like the proximity of the mesh to some other geometry).

Custom Edge Weight (cEdgeWeight) This tool assigns a set of custom values as weights for each of the edges in a MeshGraph. The set of values is remapped to the 0…1 interval.

Inputs:

(G) MeshGraph – the MeshGraph object

(W) Weights – the list of weight values to be assigned to the edges. The number of weight values needs to match the number of edges in the graph

Outputs:

(G) MeshGraph – the weighted MeshGraph object

Custom Node Weight (cNodeWeight) assigns a set of custom values as weights for each of the nodes in a MeshGraph. The set of values is remapped to the 0…1 interval.

Inputs:

(G) MeshGraph – the MeshGraph object

(W) Weights – the list of weight values to be assigned to the edges. The number of weight values needs to match the number of edges in the graph

Outputs:

(G) MeshGraph – the weighted MeshGraph object

Color Edge Weight (Color Weight) Calculates and assigns a set of values to the edges in the MeshGraph based on the brightness value averaged for each topological edge. The value is averaged based on the color values for each of the topological vertexes defining and edge. The calculated values are remapped to the 0…1 interval.

Inputs:

(G) MeshGraph – the MeshGraph object

Outputs:

(G) MeshGraph – the weighted MeshGraph object

Face Midpoint Distance Edge Weight (MDistWeight) Calculates and assigns a set of values to the edges in the MeshGraph based on the geodesic distance between the face centers of two adjacent faces. The calculated values are remapped to the 0…1 interval. The geodesic

distance is calculated for the polyline passing through the edge midpoint.
Inputs:
(G) MeshGraph – the MeshGraph object
Outputs:
(G) MeshGraph – the weighted MeshGraph object

Face Angle Edge Weight (FAWeight) Measures the dihedral angle for each topological edge of the mesh and assigns the value as weight to the corresponding graph edge. The set of values is remapped to the 0…1 interval.
Inputs:
(G) MeshGraph – the MeshGraph object
Outputs:
(G) MeshGraph – the weighted MeshGraph object

Face Size Node Weight (fsFaceWeight) Measures the area for each face of the mesh and assigns the value as weight to the corresponding graph node. The set of values is remapped to the 0…1 interval.
Inputs:
(G) MeshGraph – the MeshGraph object
Outputs:
(G) MeshGraph – the weighted MeshGraph object

3.Primary segmentation

The third section contains the workhorse tools for Ivy, the tree making algorithms. For more information on graphs and trees you can look here.

The decision to call this section primary segmentation has been taken as the result of two factors. 1. The tools in this section can partition a graph into more subgraphs while transforming it into a tree. 2. The edge removal is the basic operation in graph segmentation. This makes tree creation from a dual graph a particular case of graph partitioning. The group of tool contains four subsections with the division based on the particulars of the tree-making algorithms. We have single root graph parsing algorithms, disjoint set algorithms, multi-root graph parsing and complex graph agents.

DFS Edge Weight (dfsEdge) Depth first search graph algorithm.
A modified flavor of the standard backtracking, graph walking algorithm that uses the edge weight as guidance. The process starts in a specified node and keeps circulating over the lowest weight edges until it runs out of nodes to go. The already visited nodes are avoided. After the first journey gets stuck the process backtracks one step a time looking for unvisited nodes via the edges with lowest possible weight. The process stops when no more nodes unvisited nodes are left in the graph. This algorithm tends to produce longer strands of nodes and fewer bifurcations in the tree.
Inputs:
(G) MeshGraph – the MeshGraph object that the DFS will be used on
(S) StartFace – the starting face for the algorithm. (if no start face is present the first node in the graph will be used.
Outputs:

(G) MeshGraph – the tree MeshGraph

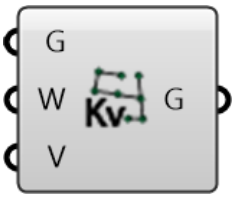
MST Prim (mstP) Minimum Spanning Tree making tool based on Prim's algorithm. This is basically a Breadth First Search BFS algorithm where each successive step is decided based on the weight landscape of the graph. The process starts from a node and grows in all directions (using all edges starting from the node) but favoring first the connections with the lowest weight. At each step, regardless of the state of the graph, the edge with the lowest weight starting from the parsed nodes and going outwards is walked. The node at the other end of the edge is added to the fold. The process stops when there are no more unvisited nodes. If the weight landscape is constant (all edges have the same weight) this algorithm equals a BFS.
Prim is a good, medium speed all-purpose algorithm.
Inputs:
(G) MeshGraph – the MeshGraph object that the DFS will be used on
(S) StartFace – the starting face for the algorithm. (if no start face is present the first node in the graph will be used.
Outputs:
(G) MeshGraph – the tree MeshGraph

MST Dijkstra (mstP) Minimum Spanning Tree making tool based on Dijkstra's algorithm. The process works in a similar way to Prim's but with one very important difference. At each step the decision is made comparing not the individual weights of the edges going out from the parsed nodes, but the weight of the lightest path of edges connecting the fringe nodes to the root (starting node). In this way a Dijkstra tree is a tree of minimal paths connecting all nodes in the graph to the starting node.
Dijkstra's is a bit slower than Prim's but a much better choice for detecting "features" in the weight landscape.
Inputs:
(G) MeshGraph – the MeshGraph object that the DFS will be used on
(S) StartFace – the starting face for the algorithm. (if no start face is present the first node in the graph will be used.
Outputs:
(G) MeshGraph – the tree MeshGraph

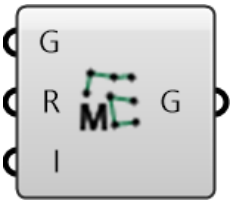
MST Kruskal (mstK) Minimum Spanning Tree making tool based on the disjoint set algorithm or Kruskal's algorithm. This tool is part of a class of algorithms that are able to parse a graph and create a minimum spanning tree without a specified starting node or root. The process starts by atomizing the graph into individual nodes. After that, at each subsequent step, individual nodes or clusters of nodes are connected, always using the edge with the lowest weight available. This, at every intermediate step produces a set of subgraphs that keep getting linked up into the final tree. By specifying a weight interval in the (W) input weights outside the interval are avoided and thus, depending on the weight landscape, a graph/mesh segmentation is achieved.
This is the fastest algorithm in Ivy and a good workhorse for any number of segmentation tasks.
Inputs:

(G) MeshGraph – the MeshGraph object
(W) Weight Limit – the interval of weights to considered
Outputs:
(G) MeshGraph – the tree MeshGraph or list of tree parts as MeshGraphs.

MST Kruskal Concavity (mstCon) A variant of the standard Kruskal's algorithm with inbuilt concavity/convexity detection. The tool works in a similar way with the standard variant but is able to detect edges that would connect regions with different concavity/convexity or flatness and increment their weight with 1 or 2. The increment is different depending on whether the jump happens from flat to convex/concave (+1) or from convex to concave (+2). This way convex, concave or flat features can be singled out and separated from other parts of the graph. The separation is controlled via the (I) input. Here an interval below 1 or 2 can be specified leaving the edges that span convex/concave/flat features outside the pick, and thus the parts hopefully disconnected.
Inputs:
(G) MeshGraph – the MeshGraph object
(I) Interval – the interval of weights to be considered
Outputs:
(G) MeshGraph – the tree MeshGraph or list of tree parts as MeshGraphs.



MST Kruskal Valence (mstKv) Another variant of the standard Kruskal's algorithm with the added possibility of specifying the preferred maximum valence for every node in the final tree graph. By specifying and integer for the (V) input the algorithm is forced to avoid adding new connections to any node if the current number of edges for the node is equal or larger than the input value. This is not an absolute rule. If the remaining nodes or clusters cannot be connected in any other way other than with a larger valence node, they will be connected. Here valence takes precedence over weight.
This is helpful in creating longer lean trees with fewer branching nodes. (For V=2) This kind of trees produce better unroll candidates.
Inputs:
(G) MeshGraph – the MeshGraph object
(W) Weight Limit – the interval of weights to be considered
(V) Valence – the maximum valence for the nodes (default = 2)
Outputs:
(G) MeshGraph – the tree MeshGraph or list of tree parts as MeshGraphs.



Multi Root MST Edge (mrMSTedge) This tool is an evolution of the weighted BFS algorithms like Prim's. In this case on the same graph a list of trees is grown at the same time. The rules are the same like in the simple single root algorithm with the addition of the fact that the individual trees cannot overlap each other. This algorithm also offers the possibility of limiting the weight that is accessible to the tree. This way the growth can be kept outside certain areas of the base MeshGraph creating a segmentation. Another segmentation is

achieved through the simultaneous development of multiple tree graphs on the same MeshGraph base. In this version of the algorithm the edge weight is considered.

Inputs:

(G) MeshGraph – the MeshGraph object

(R) RootFaces – list of start nodes for the trees

(I) Interval – the interval of weights to be considered

Outputs:

(G) MeshGraph – the tree MeshGraph or list of tree parts as MeshGraphs.

Multi Root MST Node (mrMSTnode) A similar tool with the previous one with the difference of using node weight instead of edge weight.

Inputs:

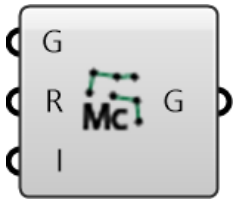
(G) MeshGraph – the MeshGraph object

(R) RootFaces – list of start nodes for the trees

(I) Interval – the interval of weights to be considered

Outputs:

(G) MeshGraph – the tree MeshGraph or list of tree parts as MeshGraphs.



Multi Root MST Concavity (mrMSTconc) Another version of the Multi Root MST Edge tool that considers concavity and convexity of the next parsed edge. Edges crossing from flat to either concave or convex regions will have the weight incremented with +1. Edges crossing from concave to convex or the other way around will have their weight incremented with +2. This way the growth can avoid those features of the MeshGraph or alternatively the edges can be left out in order to trigger a MeshGraph segmentation.

Inputs:

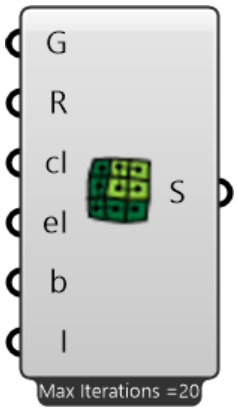
(G) MeshGraph – the MeshGraph object

(R) RootFaces – list of start nodes for the trees

(I) Interval – the interval of weights to be considered

Outputs:

(G) MeshGraph – the tree MeshGraph or list of tree parts as MeshGraphs.



Agents Control Random (AgentsCR) An evolution of the multi root minimum spanning trees tool, where each of the individual trees growing on the base MeshGraph has an individual behavior and its own limits. An individual growing tree (agent) can switch between explore and consume behaviors. The **consume** behavior equals a weighted BFS much like Prim's algorithm. Each of the fringe nodes looks for the lightest edge to add another node to the tree. The **explore** behavior is different in the sense that the same fringe nodes expand but only with one unit each and each expansion is made avoiding to touch (come into vicinity of) any other visited node. This way exploring sends out tendrils that curl or extend based on the weight landscape, while consume blankets the existing space with the visited nodes. Both behaviors prioritize small weight edges.

Each agent also possesses its own limits (the interval for edge weights that can be used) for both behaviors. This flavor of the tool switches

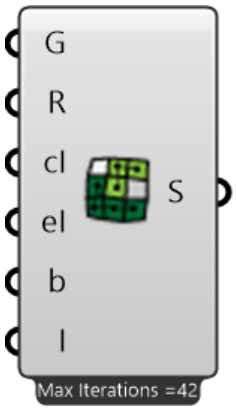
between behaviors randomly based on a coefficient set by the user. Based on that coefficient, each agent has a different chance to choose a certain behavior at any step.
Each behavior, for each agent, has a different weight interval.
This tool can potentially be used to create interlocking star-like segmentations where the parts form a pattern on the mesh.

Inputs:

(G) MeshGraph – the MeshGraph object
(R) RootFaces – list of start nodes for the trees
(cI) ConsumeInterval – the interval of weights to be considered for the consume behavior
(eI) ExploreInterval – the interval of weights to be considered for the explore behavior
(b) Behavior – a value between 0 and 1 determining the chance of picking consume or explore as a behavior for an agent at every step it takes. Values < 0.5 mean more explore and larger more consume.
(I) Iteration – the iteration to display in the viewport. The growth of every agent is cached and the user can select a certain step to see how the agents evolve. Just plug a slider and pick a number to see that intermediate state. The total number of iterations is displayed under the component.

Outputs:

(G) MeshGraph – a list of MeshGraphs the result produced by every agent.



Agents Programmed Behavior (AgentsPB) A tool very similar with the previous one but with a difference. The behavior selection is not random anymore but is selected according to a pattern of true/false values fed by the user in the (b) input. If the pattern list is shorter than the number of iterations, then the list is repeated. If the number of values exceeds the number of iterations, then the extra values are ignored.
Given a good understanding of the weight landscape this tool can be programmed to create patterns on the mesh that are both easily unfoldable and also facilitate a rapid fabrication process.

Inputs:

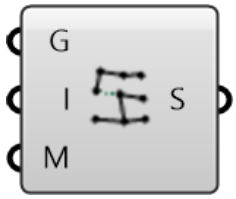
(G) MeshGraph – the MeshGraph object
(R) RootFaces – list of start nodes for the trees
(cI) ConsumeInterval – the interval of weights to be considered for the consume behavior
(eI) ExploreInterval – the interval of weights to be considered for the explore behavior
(b) Behavior – a list of Boolean values determining the behavior selected for an agent at a certain step. False = explore, True = consume
(I) Iteration – the iteration to display in the viewport. The growth of every agent is cached and the user can select a certain step to see how the agents evolve. Just plug a slider and pick a number to see that intermediate state. The total number of iterations is displayed under the component.

Outputs:

(G) MeshGraph – a list of MeshGraphs the result produced by every agent.

4. Secondary segmentation

The fourth section contains tools designed to further segment MeshGraph Trees. The tools work on the assumption that a tree graph is entirely composed of simply connected nodes and by removing a single edge, a graph segmentation is achieved. In order to use the tools in this section it is recommended that the user first transforms the dual MeshGraph of a mesh into a tree.



Weight Split Graph (WSplit) Splits a tree MeshGraph into subgraphs by deleting edges. The edges with weight inside a supplied interval are deleted. A special safeguard feature can keep graph chunks over a minimum number of nodes in order to prevent over-atomization.

Inputs:

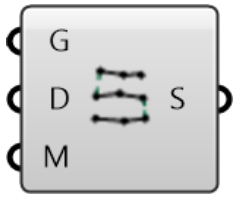
(G) MeshGraph – the MeshGraph object. Must be a tree.

(I) Interval – the interval of weights to be considered

(M) MinFaces – the minimum number of nodes/mesh faces a piece needs to have in order for the split to be validated.

Outputs:

(G) SubGraphs – the list of MeshGraph pieces.



Weight Deviation Split Graph (DevSplit) Splits a tree MeshGraph into subgraphs by deleting edges. The tree MeshGraph is parsed and the edges with a weight larger or smaller than the previous edge by more than a set amount are deleted. A special safeguard feature can keep graph chunks over a minimum number of nodes in order to prevent over-atomization.

This type of segmentation does a better job at splitting graphs at key feature points.

Inputs:

(G) MeshGraph – the MeshGraph object. Must be a tree.

(D) Deviation – the amount the edge weight needs to deviate from the previous one in order to be deleted

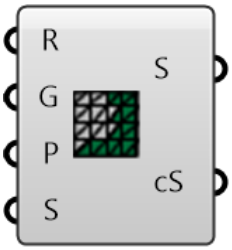
(M) MinFaces – the minimum number of nodes/mesh faces a piece needs to have in order for the split to be validated.

Outputs:

(G) SubGraphs – the list of MeshGraph pieces.

5. Iterative segmentation

This section is home to special tools that iterate over a MeshGraph and recreate segmentations based on an evolving set of variables that are read from the segmented pieces themselves. Here the result is achieved by reaching a convergence value. For now, just one tool is part of this section but more will be developed in due time.

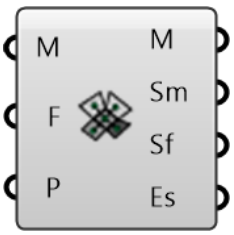


K-Means Clustering (kMeans) Splits a MeshGraph into subgraphs using the K-Means algorithm. The split is achieved iteratively by creating clusters and determining if their geometrical center converges towards a stable position. The clustering can start in two ways. The first way is the somewhat standard K-Means approach of providing the algorithm with the K number of desired clusters (input (P)) and allowing it to find start positions for them. The second approach lets the user specify exactly the start position (S) for the clusters centers thus shortening the compute time. In order to produce the iteration a timer needs to be connected to the component.

Inputs:
(R) Reset – reset switch that brings the clustering to its initial state.
(G) MeshGraph – the MeshGraph object. For best results this should be the initial dual MeshGraph so NOT a tree.
(P) Pieces – the number of pieces the algorithm should try to split the graph into. This should be used alternatively with the next input. One or the other.
(S) Seeds – manual input of the starting faces/graph nodes for the clusters. This is the alternative for specifying the number of pieces using the previous input.
Outputs:
(S) SubGraphs – the list of MeshGraph pieces.
(cS) CenterSeeds – the centers of the clusters as calculated by the algorithm.

6. Special segmentation

This section contains tools that segment the MeshGraph in order to achieve a clear end-result. In this case the segmentation can be considered a byproduct of the final tool output.

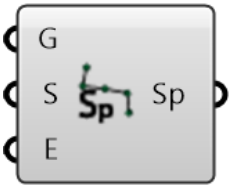


MeshGraph Unroll (mgUnroll) This tool unrolls the base mesh of a tree MeshGraph. The segmentation is the byproduct of the requirement of overlap avoidance in the unfolded state of the base mesh. The tool unrolls the tree MeshGraph and outputs a list of MeshGraphs with flat base meshes (Sf) and a list of the same segmentation but with the original mesh geometry as a base (Sm). Those two outputs are the prerequisites for the fabrication tools in the next category. The flat pieces are piled one on top of each other at the plane location provided by the (P) input. The tool also outputs the index of the edges that have to be removed in order to avoid overlaps in the flat version of the mesh.

Inputs:
(R) Reset – reset switch that brings the clustering to its initial state.
(G) MeshGraph – the MeshGraph object. For best results this should be the initial dual MeshGraph so NOT a tree.
(P) Pieces – the number of pieces the algorithm should try to split the graph into. This should be used alternatively with the next input. One or the other.
Outputs:

(M) Mesh – the unrolled mesh
(Sm) SplitMeshGraphs – the list of MeshGraph pieces on the original geometry.
(Sf) SplitFlatGraphs – the list of split flap MeshGraphs without any overlaps.
(Es) UnrollEdgeSplits – list of indexes for split edges during unroll

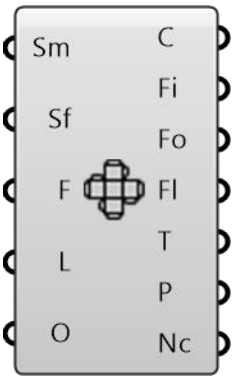
---



Shortest Paths in a Weighted MeshGRaph (sPath) An alternative use for Dijkstra's algorithm for computing the shortest (lightest) path in a weight landscape. The algorithm computes a Dijkstra tree starting from the start node until it reaches the end node. The connection between the two, inside the tree graph, is the lightest path based on the weight landscape. Multiple end point can be computed at the same time because they are part of the same minimum spanning tree.
Inputs:
(G) MeshGraph – the MeshGraph object. For best results this should be the initial dual MeshGraph so NOT a tree.
(S) StartFace – the start face (node) for the minimal path calculation.
(E) EndFace – the end face (node) for the minimal path calculation
Outputs:
(Sp) ShortestPath – a linear MeshGraph containing the path between the two nodes.

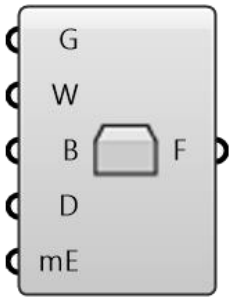
7. Fabrication

This section contains the tools that deal with the flat MeshGraphs after the unroll process. This is why the section was placed after the special segmentation section and the unroll component. The tools grouped in this section take care of the necessities for fabricating the unrolled graphs from thin flat sheets. For this scope, besides the fabrication engine the section also contains two flap making components. Those cater to both a speedy standard set-up and also an elaborate custom fabrication scenario.



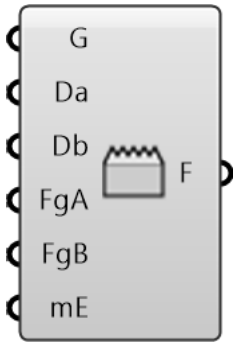
Flat Fabrication (FlatFab) The most important tool in the fabrication section works in tandem with the MeshGraph Unroll tool. The data outputted form the Unroll component is decomposed into meaningful fabrication data like curves, tag locations and tag data. The 2d fabrication data is divided into several outputs for an easy management of the cut. The component outputs separate trees of curves organized in: cuts, mountain folds, valley folds, no fold (flat) lines. Also there are separate outputs for tag text and tag locations. In order to create a useful fabrication this FlatFab component allows for the user to specify the type of flap available for each connection either between the pieces or inside every piece. The tool accepts as input the custom Flap data type that encodes all the pertinent information about the flap joinery. The next two component descriptions will detail how the flap data is encoded and how the user is able to construct and deploy flaps as part of a flat fabrication.
Inputs:
(Sm) SplitMeshGraphs – the list of MeshGraph pieces on the original geometry. This connects directly from the Unroll MeshGraph

(Sf) SplitFlatGraphs – the list of split flap MeshGraphs without any overlaps This connects directly from the Unroll MeshGraph
(F) Flaps – a list of flap data objects. The component will select from this list the appropriate flap geometry using the topological edge mesh id. This is encoded in the flap object. If a flap object with an appropriate mesh id is not present in the list, the edge will not get a flap.
(L) Location – input a standard plane here to specify the location of the processed fabrication geometry. The geometry is still piled up. This input is used just for moving the output of the component as one piece.
(O) TextOffset – a 0...1 value specifying the placement of the cut/flap edge text tags. The values determine the placement of the start point for the text along a line spanning from the face center to the edge midpoint.
Outputs:
(C) Cut Lines – a GH tree with the lines that will need to be cut in the flat fabrication. There will be a branch for each MeshGraph piece with a sub-branch for each face. On each sub-branch a list with all the cut lines for that mesh face will be grouped.
(Fi) FoldIn Lines – another GH tree with the lines that will be folded in to construct the original geometry. The branch set-up is similar with the Cut output.
(Fo) FoldOut Lines – a similar GH tree but this time for the lines that will have to be folded out. Similar branch set-up
(Fl) Flat Line – GH tree with the same organization but containing the lines that will not need to be folded or cut. These lines are usually between the coplanar adjacent faces. The contents of this output is usually not needed for fabrication.
(T) Text – the text content for each of the tags that will accompany the flat fab lines.
(P) Place Holders – the location data needed for the text tags.
(Nc) Node Curves – the geometry added to the node in order to be unrolled with the MeshGraph.



Simple Flap (sFlp) This tool creates a simple beveled flap for standard glue connections in paper models. The tool encodes a flap data type that will be used by the FlatFab component. Each simple flap needs to know the topological edge it will be applied to. In order to produce flaps for the whole model the tool needs a list with all the edges from MeshGraph that will be cut. Ivy has a tool that does just that it is called EdgeTypes. Alternatively, a list with indexes for all the edges in a mesh can be fed in the (mE) input. Even if more flaps objects will be created the FlatFab component will only use the ones that are required. This kind of approach was chosen because it allows to combine flaps with different settings for the same MeshGraph.
Inputs:
(G) MeshGraph – the MeshGraph object. The original MeshGraph before any segmentation.
(W) Width – the width of the glue flap.

(B) Bevel – the bevel of the flap. This is distance based. A bevel distance equal to the width will produce ca 45-degree bevel.
(D) Double – this boolean input specifies if the flap is double. A double flap will produce a 2d cut flap on both cut edges resulted from a split topological edge or MeshGraph edge. This is useful for creating flaps that remain perpendicular to the geometry after gluing.
(mE) Mesh Edge Index – the mesh topological index that will be bound to the flap.
Outputs:
(F) Flaps – a list of flap data objects for the FlatFab.

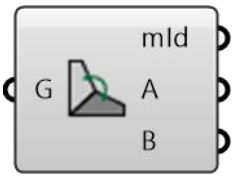


Custom Flap (cFlp) This tool is a generalization of the previous one and creates flaps giving the user complete control over the geometry and use of the flap. The tool is able to create custom geometry different for each side of a double flap and reads flaps constructed on the original non-flat geometry. In this way the user is able to create a visualization of the flaps on the original geometry and thus gauge the effect that fabrication will have on the final product. The same general construction lines are followed just like in the case of the simple standard flap. In order to be properly identified by the FlatFab component the flap object needs to encode the edge index it belongs to. Also the original MeshGraph is needed in order to properly calibrate the geometry in its flat state. The custom flaps can be constructed at any angle on the original geometry in order to closely simulate the fabricated product. The custom flap component needs to read those custom directions for each flap in order to facilitate a proper transfer of the geometry towards the flat state.
In order to facilitate complex planar geometries for each side of the flap the geometry inputs for the two sides are lists. This ensures that flaps constructed from multiple curves are properly translated to the flat state and matched to their respective mesh ids. This approach requires a careful data set-up for the component inputs. Basically it is recommended (in the case of multiple flaps) to graft all inputs with 1 element per branch except the geometry inputs (FgA, FgB). The geometry inputs will have one or more elements per list. This is the most effective way of matching one mesh Id (mE), and custom directions (Da, Db) with multiple curves (FgA, FgB) for a series of flaps. Please consult the examples supplied with Ivy 0.8 for more information.
Inputs:
(G) MeshGraph – the MeshGraph object. The original MeshGraph before any segmentation.
(Da) Direction a – the custom direction set up on the original geometry for the side A of the flap. This input is optional.
(Db) Direction b – the custom direction set up on the original geometry for the side B of the flap. This input is optional.
(FgA) Flap Geometry A – this is a list input containing all the curve entities needed to properly draw the flap's side A. This allows for designs with holes or cuts.
(FgB) Flap Geometry B – this is a list input containing all the curve entities needed to properly draw the flap's side B.

Outputs:
(F) Flaps – a list of flap data objects for the FlatFab.

8. Mesh Info

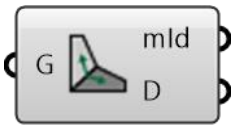
This section contains a set of tools that extract information from MeshGraphs. They are either tools that are not ready available in Grasshopper, or special facilitators that inform important MeshGraph tools and processes.



Edge Angle (eAngle) This component extracts the dihedral angle for each topological 2-manifold edge inside an MeshGraph. The angle is calculated from the base mesh of the graph and is outputted in radians. Because Rhino outputs only the smallest angle between the faces the component also outputs the "bend" convex/concave for the edge. This tool is designed to provide primary information for the Custom Edge Weight tool for the scenarios where the angle number is not used directly as raw data for the weight.

Inputs:
(G) MeshGraph – the MeshGraph object. For best results this should be the initial dual MeshGraph so NOT a tree.

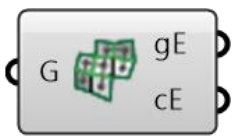
Outputs:
(mId) Mesh Id – the mesh id that will accompany the angle value
(A) Angle – the angle value in radians.
(B) BendType – 1 = flat; 2 = concave; 3 = convex;



Edge Traversal (eTrav) A similar tool with the previous one designed to work in a much the same scenario. This component calculates the geodesic distance between the centers of two adjacent faces going through the common edge's midpoint.

Inputs:
(G) MeshGraph – the MeshGraph object. For best results this should be the initial dual MeshGraph so NOT a tree.

Outputs:
(mId) Mesh Id – the mesh id that will accompany the distance value
(A) Distance – the distance value.



Edge Types (eTypes) This component extracts the edges from a base mesh by their MeshGraph statute. The edges can be either a graph edge (one connecting nodes inside the graph) or a cut edge (one that was deleted from the dual graph in the segmentation/tree making process)

Inputs:
(G) MeshGraph – the MeshGraph object.

Outputs:
(gE) GraphEdges – mesh Ids for the graph edges
(cE) CutEdges – mesh Ids for the cut edges

Orange Peel Edges (OPE) The tool creates a pattern of nodes that develop concentrically radiating from a set of mesh features. The features are defined by a tree of mesh vertex ids provided by the user. The vertex ids are grouped in branches each defining a feature that ripples its own "waves". The component outputs the edge ids for the edges that span between the waves. This is useful in order to assign weight to those edges and thus facilitate a segmentation in concentric strips.

Inputs:

(G) MeshGraph – the MeshGraph object. For best results this should be the initial dual MeshGraph so NOT a tree.

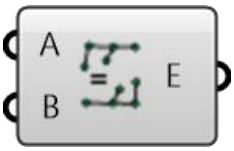
Outputs:

(gE) GraphEdges – mesh Ids for the graph edges

(S) StartVertices – the vertices defining the features of the mesh. The centers of the ripples. In order to separate the features, the lists of vertex ids need to be separated on different branches.

9. Other tools

This section contains miscellaneous tools that didn't find a place in the other sections. They are not used too often in the typical Ivy workflow, but might prove useful on occasion.



Graph Equality (GraphEqual) This component test two graphs for equality. Equality for two graphs means the nodes and edges are equal

Inputs:

(A) MeshGraph A – the first MeshGraph object.

(B) MeshGraph B – the second MeshGraph object.

Outputs:

(E) Equality – a boolean value attesting the equality of the two graphs

---

Cull Graph Duplicates (CullGrph) A tool that uses the functionality of the previous tool to get rid of the graph duplicates from a list.

Inputs:

(L) MeshGraph List – the list of graphs with duplicates

Outputs:

(L) MeshGraph List – the list of graphs without duplicates

Get Deepest Nodes (DeepestN) The tool computes the distance (in graph steps) of each node in a tree MeshGraph to the nearest leaf (a node with only one edge connection). It returns the index for the node(s) with the largest number of steps to a leaf, those are the deepest nodes in a tree.

Inputs:

(G) MeshGraph – the tree MeshGraph object.

Outputs:

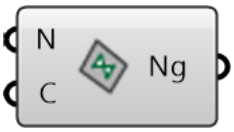
(N) NodeList – the list of deepest nodes

Set Tree Root (TreeRoot) This tool sets the root of a MeshGraph to an arbitrary node.

Inputs:

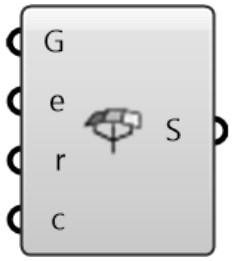
(G) MeshGraph – the MeshGraph object.

(R) RootIndex – the mesh Id for the node that will become the new root.
Outputs:
(G) MeshGraph – the MeshGraph object with the new root.

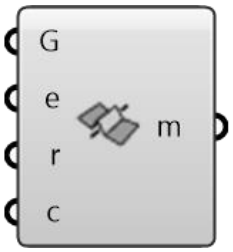


Set Node Geometry This tool creates a node geometry object encoding a number of flat curves that will be unrolled together with the MeshGraph and the corresponding mesh.
Inputs:
(N) Node identifier – this is the node mesh id that will identify the node in the MeshGraph.
(C) Curves – the curve(s) that will inhabit the node. These need to be nested in the corresponding mesh face and in the same plane.
Outputs:
(Ng) Node Geometry – a node data object shell with a node id and the corresponding curve geometry.

Node Geo to Graph This tool inserts a node geometry object into a MeshGraph.
Inputs:
(G) MeshGraph – the MeshGraph object.
(Ng) Node Geometry – a node data object shell with a node id and the corresponding curve geometry.
Outputs:
(G) MeshGraph – the MeshGraph object with the added node geometry.



Graph Structure (grphStruct) This tool makes use of the tree mesh graph to create a structure that follows the geometry of the graph and is offset-able with custom distances from root to leafs.
Inputs:
(G) MeshGraph – the tree MeshGraph object.
(e) EndHeight – the distance of the structure from the mesh at the leaf level
(r) RootHeight – the distance of the structure from the mesh at the root level
(c) Coefficient – a coefficient for the height for each step along the graph. This will be multiplied with the distance from the base mesh at each step thus decreasing or increasing the distance (if below 1.00 or above 1.00)



Crease Mesh (creaseM) This tool makes use of the tree mesh graph to create a new mesh with creases that follow the geometry of the graph. The depth of the creases is user controllable and it follows the flow of the graph like a hydrographic basin.
Inputs:
(G) MeshGraph – the tree MeshGraph object.
(e) EndHeight – the depth or height of the creases at the leaf level
(r) RootHeight – the depth or height of the creases at the root level

(c) Coefficient – a coefficient for the height for each step along the graph. This will be multiplied with the distance from the base mesh at each step thus decreasing or increasing the distance (if below 1.00 or above 1.00)

0. Index of Ivy commands.