0.1 Minimal spanning tree

Scene: *Euclidean spanning tree*

The scene shows a collection of poits that are called *sites*. Our goal is to connect them in the most efficient by links directly connecting pairs of sites. The cost of the interconnections is the sum of costs of all links. In this scene the cost of a link is equal to the Euclidean distance of its end-points (some other scenes use a different metrics).

Let us recall that the Euclidean metric (denoted also as L_2-metric) defines the distance d of two sites $s = [s_x, s_y]$ and $t = [t_x, t_y]$ as follows:

$$d = \sqrt{(s_x - t_x)^2 + (s_y - t_y)^2}.$$

Click the button [**Step**], Algovision displays the optimum interconnection of sites. [**Back**] erases the solution.

The set of sites can be modified if you switch from the choice [**Compute**] to [**Edit**]. When in the edit mode, if you click to a point that doesn't belong to any site, a new site is created. Sites can be dragged by the mouse. If you click to an existing site with the right mouse button pressed, Algovision suggests deleting the site, another click to the suggesting rectangle executes the operation. After switching back to the compute mode, the minimum spanning tree of a new configuration can be computed.

Try to modify the site set, and observe how the optimal tree changes.

Scene: *A spanning tree in a metric graph*

The scene shows again a site set, but some of site pairs are connected by blue links. The goal is again to find the cheapest interconnecting all sites together (it will be drawn in green), but under the constraint that any interconnecting link must be selected among pairs of sites that are connected by a blue link.

Of course, the graph given by the blue links must be connected, because any subgraph of a disconnected graph is disconnected as well. If the graph is disconnected, Algovision protests and you have to switch into the edit mode and make the graph connected.

A spanning tree computed in the previous scene is a special case when the constraint graph is complete (all pairs of vertices are connected by a blue edge).

Click to [**Step**] to display the solution, click to [**Reset**] to come back to the original input.

When in the edit mode, the set of sites can be edited in the same way as in the previous scene. If a site is deleted, Algovision deletes all blue links incident with the site as well.

Moreover, it is possible to connect two directly unconnected sites by clicking the mouse over one site and subsequent clicking over another site (the cursor drags the free end of an edge being added). The editor adds a new link, and starts immediately a new process of adding further link from the latter site. The process can be stopped by clicking to the empty space.

Finally, when a blue link is clicked by the right button, Algovision suggests deleting the link and executes the operation if the menu is clicked.

Scene: *A spanning tree in a labeled graph*

The scene is similar to the previous one, but the cost of a link is not given by the Euclidean distance of its end-sites but it is given by a (positive) integer written next to the link.

When in the edit mode, and a new blue link is created, its cost is set randomly between 1 and 99. It is also possible to change the cost of a blue link by clicking it by the right button of the mouse and choosing [**Set length**]. A number field will appear in the control bar and you can change the cost of the selected link in the field. Zero or negative cost of an edge is not allowed, because this would change the nature of the problem. We restrict ourselves to small integer costs just because this simplifies graphics, but all algorithms work in the same way for arbitrary positive real cost.

Play freely with the site set, links and costs to observe how they influence the corresponding optimum spanning tree.

Scene: *A general algorithmic scheme*

This scene shows a general scheme to determine the minimum spanning tree. The scheme allows large freedom in selection that occurs in odd steps, while being deterministic in even steps. Particular algorithms restrict odd step freedom, usually in order to obtain faster computation. Of course, any implementation of the scheme guarantees the correct answer.

In this scene we assume metric costs of links, the case of general costs will be dealt with in the next scene.

The site set can be modified as before, but the change resets the computation if it is not completely finished.

The algorithm scheme is quite simple: links are added into an originally link-free graph one-by-one in such a way that no cycle is created.

As mentioned, at the beginning the graph contains all sites but no links. During the computation certain links are present, but the graph contains no cycles. This means that the graph is a *forrest* the components of which are unrooted trees. In this chapter, we will use the term "forrest" to denote the forrest generated by edges that were already put into an unfinished spanning tree.

Each link addition decreases the number of components by 1 by merging two components into one. It is clear that no link can be added to connect sites that are already in the same component of the forrest. The computation ends when there is just one component.

In order to identify components we use colors. Nodes of any component have the same color, which is different from components of other components. Especially, any two sites have different color at the beginning.

Since there are N one-site components at the beginning and 1 component at the end, the computation consists of $N-1$ repetitions of the following two steps:

Step: *A component choice*

Choose an *arbitrary* component $\mathcal{C}$ of the forrest.

Step: *Link choice*

Among pairs (s,t) of sites such that s belongs to the component $\mathcal{C}$ and t is outside of $\mathcal{C}$ choose the one with the smallest cost, breaking ties arbitrarily, and add it into the forrest.

The choice of a component is absolutly free, on the other hand the link choice is uniquely determined by the component choice (unless there are more links of the minimum cost).

In the present scene Algovision selects a component in the component choice step randomly.

In order to illustrate the selection of the minimum link the selection process is animated by extending neighborhoods of the same increasing radius around all nodes of the selected component $\mathcal{C}$ until the first neighborhood touches a vertex outside of $\mathcal{C}$, which signals that the shortest link was found. The next click of the button **[Step]** connects the optimum pair of sites.

Scene: L_1 *metric*

The scene repeats the previous one, but the cost d of a link connecting two sites $s=[s_x,s_y]$ and $t=[t_x,t_y]$ is not determined by the Euclidean metric, but by L_1-metric (or the *sum* metric) defined as follows:

$$d=|s_x-t_y|+|s_y-t_y|.$$

Neighborhood of a site is not a circle, but a square with corners on the horizontal and vertical lines passing through the site.

Scene: L_{max} *metric*

The scene repeats the previous two, but the cost d of a link connecting two sites $s = [s_x, s_y]$ and $t = [t_x, t_y]$ is not determined by the Euclidean metric or L_1-metric, but by L_{max}-metric (or L_∞-metric or the *maximum* metric) defined as follows:

$$d = \max(|s_x - t_y|, |s_y - t_y|).$$

Neighborhood of a site is not a circle, but a square with horizontal and vertical sides.

Scene: *Metric graph*

The scene repeats the previous ones, the cost of a link is again its Euclidean length, but the interconnection links must be chosen among blue links in the same way as in the second scene of this applet.

Neighboorhoods of sites are not shown completely, we just show how far they reach in the blue links. The closest neighbor of a chosen component is found when the pink part of some (originally) blue link reaches a site that is outside of the selected component.

Scene: *Labeled graph*

The scene repeats the previous one, the interconnection links must be chosen among blue links, and the cost of a blue link is given by the number next to the link in the same way as in the third scene of this applet.

We use again pink segments of links to show neighborhoods. However, in this case the speed of the increase of segments is not the same, but correspond in the obvious way to the cost of the link. (More precisely, the length of the pink segment in a link of the cost c is k/c, where the value of k is the same for all links of the graph and increases in time).

Scene: *Algorithm Jarník - Prim*

The scene shows an algorithm described by Jarník in 1930 and rediscovered by Prim in 1957, which is a simple variant of the general scheme. At the beginning choose a node v_0 arbitrarily (Algovision does it at random) and later the component $\mathcal{C}$ of the general scheme is always chosen as the one containing the node v_0.

During the computation one component always grows (the one containint v_0), while the other components remain one-node and are "eaten" by the large component.

The scene offers full choice of variants. There are three basic choices: [**Valued graph**] offers blue link constraints with general numerical costs, [**Metric graph**] has blue link constraints, but their costs are determined by a metric, and finally [**Complete graph**] has no constraints, and the cost of a site pair is again determined by a metric. Note that the latter possibility is equivalent to the constrained metric graph, where all site pairs are connected by a blue link (a complete graph). In the second and third cases, L_2, L_1 and L_{max} metrics can be chosen. Of course, it is also possible to change the site set and, if appropriate, blue links and their costs.

An efficient implementation of Jarník - Prim algorithm requires an efficient choice of the minimal link starting in the large component. This can be done advantageously by using a priority queue. Once a new node v is inserted into the large component, all links connecting v with nodes of the large component are removed from the priority queue, because they no more represent links connecting the large component with an outside vertex, on the other hand all remaining links strating in v are addeed into the queue, because they are now links starting in the large component (in v) and ending elsewhere.

Scene: *Kruskal*

This scene shows another minimum spanning tree algorithm that was described by Kruskal. At the beginning it might seem that the Kruskal algorithm is not a special case of the general scheme, and it is also visuaized in a different way. However, we will show that it can also be considered as an implementation of the general scheme.

The algorithm again works by including links to connect components of the forrest that will eventually become a spanning tree. It works as follows: links of the graph are sorted by non-decreasing costs. Then, the links are scanned in this order. For each link the algorithm checks if it connects two sites in different components or two sites that belong to the same component. In the former case the link is added to the graph, while in the latter case it is simply discarded.

If small steps are chosen in the control bar, the considered edge is first marked using the red color, if it is added into the forrest, it gets the original color and the green underground in the next step, otherwise it changes the color to yellow. If longer steps are chosen, the red-marking step is skipped. If **[Computation]** is selected, one gets immediately the result without giving intermediate steps.

In other words, at each moment we add the link that has the minimum cost among all links that connect two different components of the unfinished spanning tree.

The computation can be observed for a graph with blue link constraints with numerical or metric costs. All three metrics L_2, L_1 and L_{max} are available. There are too many links in the case of the complete graph, and hence this variant (explained in the previous scene) is not implemented.

Algovision considers links in the order of non-decreasing costs. If a considered link connects two different components of the forrest, it is added into the collection of links selected for interconnection (it is emphasized by the green color). A link that connects two sites that are in the same component at the moment it is explored is simply discarded (recolored to red).

Scene: *Kruskal again*

This scene shows another implementation and visualization of Kruskal algorithm. For simplicity only unconstrained collection of sites is considered (no blue links), all three metrics L_2, L_1 and L_{max} are available.

The implementation is simple: we again add edges one-by-one into originally empty forrest until it eventually becomes a tree completely connecting all sites. At each step we select a link of the minimum cost among link connecting sites from different components.

The implementation is visualized as follows: we draw neiborhoods of the the same radius increasing with the time around *all* nodes of the graph. The shape of neighborhoods is again determined by the selected metric. The neighboorhoods have the same color as their centers, which make it possible to distinguish neighborhoods of nodes from different components. Touch or overlay of neighborhoods of nodes of the same component is not interesting, because an optional link connecting such nodes would not be inserted into the forrest tree.

However, when two neighborhoods of different colors meet for the first time, the shortest link connecting two different components is found. The link is then added into the forrest, and the same procedure begins again, until the forrest is fully connected.

Try computation according Kruskal algorithm for different site sets and metrics. The algorithm does not give any freedom (except for the case when two or more links of the *same* cost are found to connect different components - in such a case Algovision chooses one of the links at random without asking the user).

Kruskal algorithm is also an implementation of the above-described general scheme: the choice of a component is done as follows: the link e of the minimum cost among links connecting different components is found and then any one of the two components that are bridged by the link is chosen as the component $\mathcal{C}$. Then, of course, the choice of the minimum cost link connecting $\mathcal{C}$ with other components must (or can, in the case of tie) return the link e, and therefore this implementation of the general scheme computes in the same way as the Kruskal algorithm.

An efficient impementation of Kruskal algorithm requires initial sorting of links according their costs and then repeated decision if a link connects two different components. The former problem is easily solved by selecting one of sorting algorithms, e.g. those that were described in this book, while the latter one is typically solved by using union-find data structure that was explained in the chapter on data structures.

Scene: *Correctness*

This scene shows why any implementation the general scheme is *always* correct, i.e., the obtained spanning tree has the smallest possible sum of link costs among all spanning trees satisfying the constraints. The key observation is to show that during the whole computation the following statement is satisfied:

I: there is at least one minimum spanning tree containing all links of the forrest built according the general scheme.

The statement (I) obviously holds at the beginning for trivial reasons, because at that time the forrest contains no links. If (I) holds in the moment when the forrest becomes a spanning tree, that the minimum spanning tree containing the constructed tree must be equal to it, i.e. the constructed tree would be minimal.

It is therefore sufficient to show that adding any link chosen according the general scheme does not violate the statement (I). In order to prove it, we will add additional steps into the computation in a way that would not alter the result of computation. The added steps are used only to explain logical relations that hold during the computation.

The new link becomes red and retains the color during additional steps, and hence its color is different from colors of older links of the forrest. We will show what would happen if the red link addition obstructs (I), i.e., if no minimum spanning tree containing old links of the forrest contains the red link. Moreover, colors of the components that are bridged together by the red link remain unchanged during the additional steps, even though the red link merges them together. Recoloring occurs only after additional steps are finished.

The additional steps are following:

Step: *Selected component marked*

The component that was selected as the component $\mathcal{C}$ of the general scheme is highlighted by white background of its nodes.

Step: *Alternative spanning tree*

As we suppose that the condition (I) was valid, but became invalid after addition of the red link, there would exist a minimum spanning tree $\mathcal{T}$ that contains all old links of the forrest, but no such tree contains the red link.

In this step a spanning tree $\mathcal{T}$ is shown that contains all old forest links (green); the other links of $\mathcal{T}$ are yellow, but the red link doesn't belong to $\mathcal{T}$. [1].

Step: *Alternative path*

As we assume that the green-yellow spanning tree $\mathcal{T}$ is connected, the end-nodes of the red link must be connected by a path π in $\mathcal{T}$. Since the red link does not belong to $\mathcal{T}$, it is also outside of π. The path π is marked by the magenta background of its links.

Step: *Bridge*

The red link starts in the component $\mathcal{C}$ and finishes outside of $\mathcal{C}$. The same would be true for the path π. Let (u, v) is the first link of π such that u belongs to $\mathcal{C}$ and v is outside of $\mathcal{C}$. We would refer to the link (u, v) as to a *bridge*. The bridge becomes black in this step. If there are links of the path π preceeding the bridge in the path, they change their color from light green to dark green.

Step: *Better tree*

A black bridge has one end-node is the component $\mathcal{C}$ (sites with white background) and the other end-node outside of $\mathcal{C}$. However, the same is true for the red link, and the red link was selected in such a way that is had the smallest cost among such links. Consequently, the cost of the black bridge is as least as large as the cost of the red link.

[1] In fact, $\mathcal{T}$ is not minimal; our assumtion of invalidity of (I) is incorrect, a minimum spanning tree containing the old eddges *must* contain the red link as well in most cases - what is shown is the minimum spanning tree among those that contain old links but not the new red one

In this step the bridge is removed from the paths π and from the tree $\mathcal{T}$, and the red link is added into $\mathcal{T}$ to get a new graph $\mathcal{T}'$. The new graph is again a spanning tree: the only cycle that the red link closed with $\mathcal{T}$ was the cycle consisting of the red link and links of the path π with the magenta background; the cycle was destroyed by removing the black bridge that was one of the links of the tree $\mathcal{T}$. On the other hand, the new graph $\mathcal{T}'$ is connected; what was earlier connected through the black bridge is now connected by means of the remaining links of the path π and the red link.

Therefore the graph $\mathcal{T}'$ is a spanning tree, and because it was obtained from $\mathcal{T}$ by removing the bridge and adding the red link that has cost that is smaller or in the worst case equal to the cost of the bridge, the cost of the new spanning tree is at most the cost of $\mathcal{T}$. However, we supposed that the tree $\mathcal{T}$ was the minimum spanning tree, and hence the cost of both $\mathcal{T}$ and $\mathcal{T}'$ are the same, which means that $\mathcal{T}'$ is also a minimum spanning tree - a contradiction with the original assumption.