Agg Example: Finding The Shortest Path

Claudia Ermel
TU Berlin
e-mail: {lieske}@cs.tu-berlin.de

We present a small, but non-trivial sample graph grammar to illustrate the features of the Agg system and the advantages of visual programming. For this purpose, we choose the well-known "shortest path" problem which is also an essential part of the tour planning in shipping companies. Search procedures are ubiquitous, popping up whenever alternatives have to be explored to solve a problem. Here, we illustrate a route finding procedure. Given a map (in our example, the map shows some cities, connected via highways), the procedure finds the shortest path from a start position (a certain city) to a goal position (another city).
The most obvious way to find a solution is to devise a bookkeeping scheme that allows orderly exploration of all possible paths. It is worthwhile to note that the bookkeeping scheme must not allow itself to cycle in the graph. It would be senseless to go through a sequence like *Start* - *City*1 - *Start* -*City*1 ... over and over again. With cyclic paths terminated, graphs are equivalent to trees. Therefore, we will generate the search tree corresponding to our road map graph. Drawing the children of one tree node is called *expanding* the node. Tree nodes are said to be *open* until they are expanded, whereupon they become *closed*. The *length* of a path in the tree is given by adding the distances between the path nodes.
To solve our problem we have to look for the shortest path in our search tree from the root node to the goal node. We define the problem by drawing the graph to be explored as the start graph of our graph grammar. Nodes in the start graph are cities (type *City*), a directed edge models the highway connection between two cities, the length of a highway is encoded as edge attribute. Of course, there always have to be two edges with opposite directions and the same distance value between two cities to denote that highways can be used in both directions. The left picture of figure 1 shows an example road map as typed, directed, attributed graph, drawn with the Agg graph editor. Directions of edges are denoted by colours in Agg: The edge part connected to the source node is drawn black, the part connected to the target node has a lighter colour. Labelling *City* nodes is realized by assigning values to the `String` attribute `n`.
We use the *Branch-and-Bound* algorithm, as described in [Win84] that builds a search tree by tracing through possible paths beginning at the start node, compares the length of the partial paths already found and expands the open node at the end of the shortest partial path. Expansion continues until there is a path reaching the goal that is of length equal to or shorter than all incomplete paths terminating at open nodes (leaves).
Let us consider the example in figure 1 and assume we want to find the shortest path from the start node labelled "S'' to the goal node labelled "G''. Looking only at the first level of the corresponding search tree on the right-hand side of figure 1, the distance from

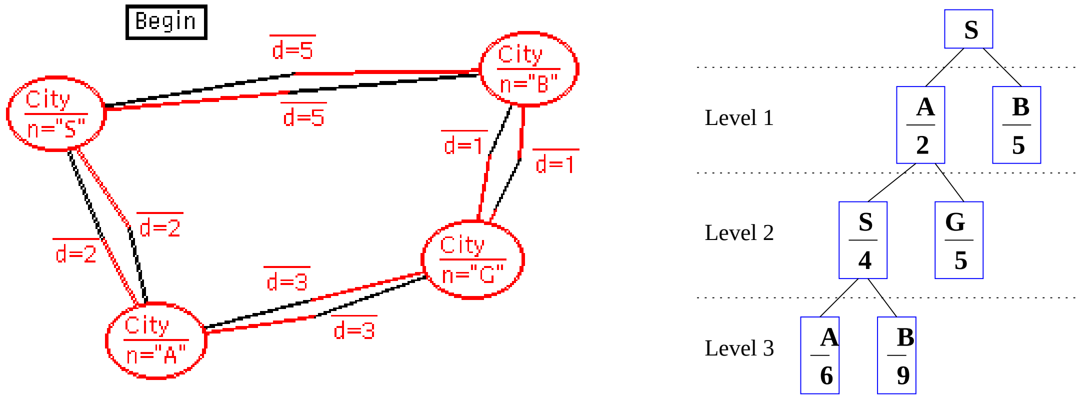


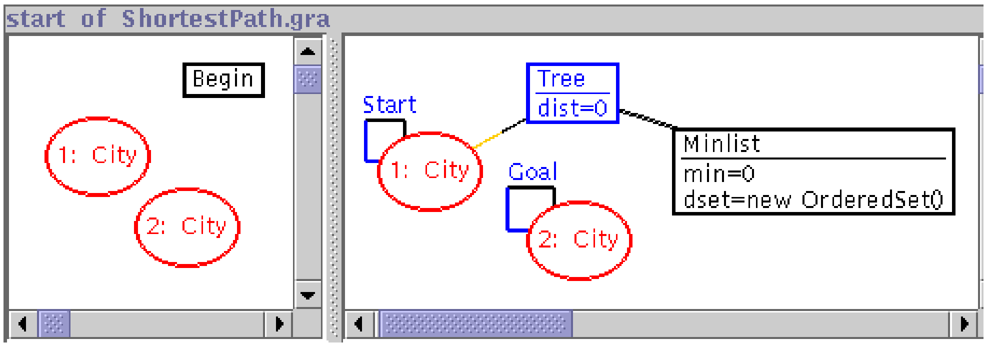
Figure 1: The Branch-and-Bound Algorithm

S to node A is clearly less than the distance to B. Expanding A at level 2 leads to a path containing the goal node. But it reveals at the same time another open path with a shorter length (so far), namely the way back to node S. Therefore, this node has to be expanded once more, leading to the two open paths in level 3. Here we see that there is no point in calculating the path length for any of the three open paths through the nodes A or B since any of the path's length is at least 5 already, and hence longer than the path to G for the known solution through A.

We see, the algorithm terminates in spite of circles in the original graph. Optimizations of the basic branch-and-bound search algorithm are given in [Win84]. Here, we are interested in demonstrating an intuitive implementation of the basic algorithm using attributed graph grammars in the AGG environment.

The road map shown in figure 1 is the start graph of our graph grammar. The rules to be applied to the working graph realize the branch-and-bound algorithm. Each rule is applied as often as possible, transforming the working graph "in place". If a rule is not applicable any more, the next rule is tried. Matches (i.e. mappings from the left-hand side to the working graph) are computed automatically or defined by the user.

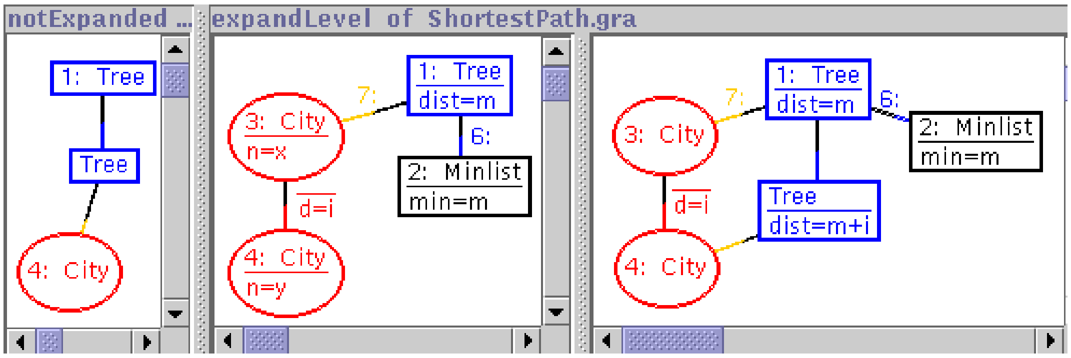
Our graph grammar *ShortestPath.gra* consists of the following six rules:



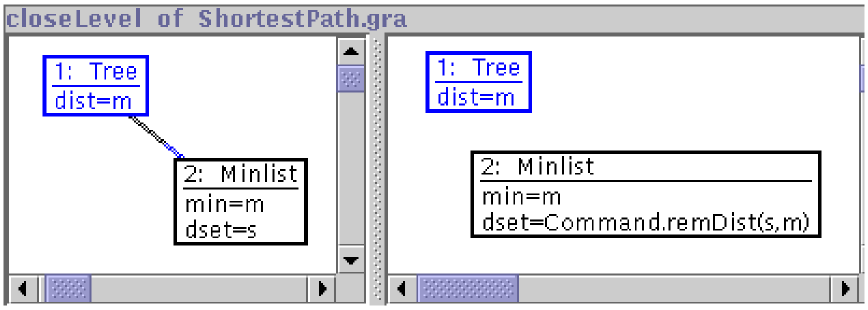
1. Before applying the *start* rule, the user identifies the start and goal nodes of the connection he wants to know the shortest traveling length of by defining a partial match that is, by mapping the two $City$ nodes of the left-hand rule side to two $City$

nodes of his choice in the search graph. Rule and match morphisms are denoted in AGG by equal numbers for each mapping. Thus, the *City* node with the number 2 will be marked by a *Start* loop, whereas *City* node no. 3 is defined as the goal node of the search procedure.

The rule generates the root of the search tree and the initial bookkeeping node (type *Minlist*) that contains a list of the current path lengths and stores the current minimum value after each expansion step. Tree nodes are connected by edges with their corresponding *City* nodes. The deletion of the *Begin* node makes sure that the *start* rule can be applied to the start graph only once.



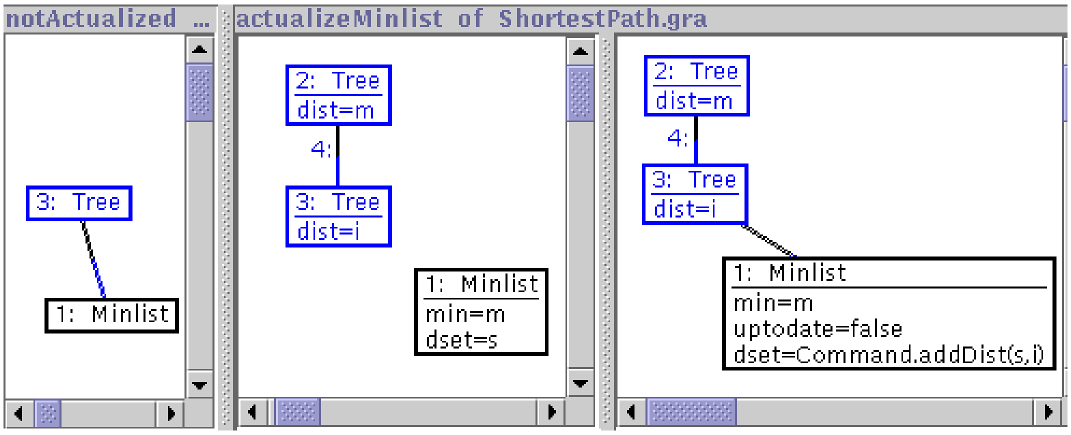
2. The *expandLevel* rule expands a node with the current minimum value that is, it generates the next level of the search tree. The negative application condition (short NAC, shown left to the left-hand rule side) ensures that no child is generated twice. The (partial) mappings from the left-hand rule side to the NAC are again shown via equal numbers. This rule has to be applied as often as possible, until all children of all nodes with the current minimum value have been found.



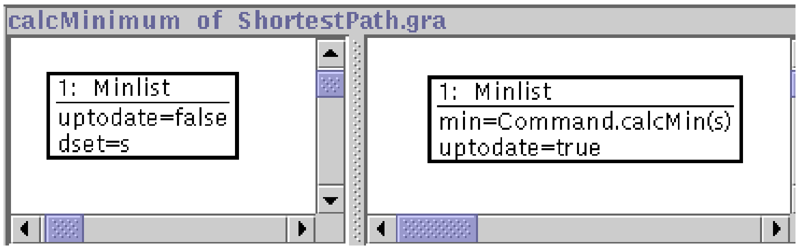
3. The rule *closeLevel* deletes the connection between the *Minlist* node and the previously expanded node as only values about paths to open nodes are stored, and the previously expanded node is now closed. At the same time, the list entry of the path to the previously expanded node is deleted from the distance list `dset` at the *Minlist* node. The deletion is realized by a call to the Java `OrderedSet` class method `s;remove` from the Java library `com.objectspace.jgl`[1].

   When the rule *closeLevel* is applied as often as possible, all previously expanded nodes are closed.

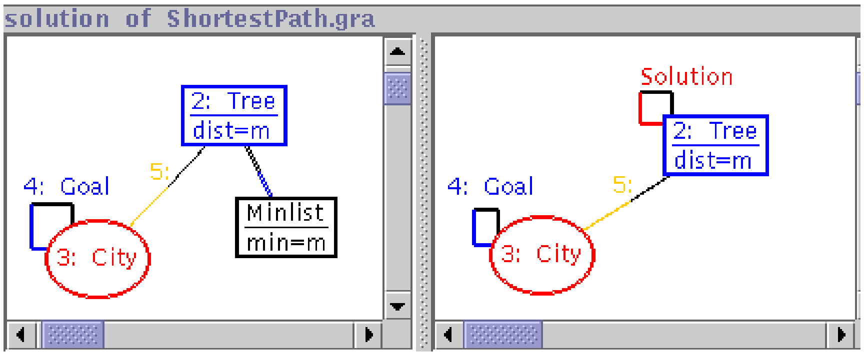
[1] Java Generic Collection Library



4. The rule *actualizeMinlist* generates an edge between the *Minlist* node and each new leaf node in the tree that has just been generated by the *expandLevel* rule. The NAC ensures that only one edge will be generated between the *Minlist* node and each new leaf node. The distances between the start node and the new leaf nodes are stored in the *Minlist* attribute `dset` containing all distances of partial paths. Again, a Java `OrderedSet` class method is called to add distances to `dset`, namely the method `s;add`.
   Setting the *Minlist* attribute `uptodate` to `false` means that the computation of the minimum of the values in the distance list is now out of date. The correct minimum will be computed in the next rule *calcMinimum*. The rule *actualizeMinlist* has to be applied as often as possible, until the *Minlist* node is connected to all leaves in the search tree.



5. Applying the rule *calcMinimum*, the current minimum of all open paths, stored in the *Minlist* attribute `min` is actualized accordingly by calling the Java method `calcMin(OrderedSet s)` from the user-defined class `Command` which is shown in figure 2. To ensure that the rule can be applied only once, right after the actualization of the distance list, the attribute `uptodate` is set to true again after one rule application.



6. The *solution* rule realizes the termination condition: If the goal node is reached in a current path *and* the distance from the start node to the goal node is equal to the minimum stored in the *Minlist* node attribute `min`, then the minimum is the solution, and the path in the search tree from the start node to the goal node represents the shortest connection between the two *City* nodes in the graph. The solution goal node in the search tree is marked by a *Solution* loop. This rule can be applied only once, because the *Minlist* node is removed when the solution is found.

```
import java.util.*;
import com.objectspace.jgl.*;

public class Command

{
 public static OrderedSet addDist(OrderedSet s, int i)
 {s.add(new Integer(i));
  return s;
 }
 public static OrderedSet remDist(OrderedSet s, int i)
 {s.remove(new Integer(i));
  return s;
 }
 public static int calcMin(OrderedSet s)
 {Enumeration e=s.elements();
  int min = ((Integer)e.nextElement()).intValue();
  return min;
 }
}
```

Figure 2: User-Defined Java Class Command.java

If the *solution* rule cannot be applied (because no path containing the goal node is yet found, or because the distance to the goal is longer than the length of other partial paths), the search must be continued by jumping back to the *expandLevel* rule. This jump is realized in AGG by starting the graph transformation again with the current working graph. As the start rule is applicable only once (because of the *Begin* node being there only in the beginning), we then actually realize a loop over all other rules that is, a jump back to the *expandLevel* rule.

Figure 3 shows the rule application strategy just described. The application mode (*) means *apply as often as possible*. For three rules, we actually realize the mode (1) meaning *applicable only once*.

Rule name	*Application*	*Description*
start	**(*) = (1)**	initializes search tree
expandLevel	**(*)**	generates all leaves for all open minimal paths
closeLevel	**(*)**	deletes old min. value and disconnects Minlist from all closed nodes
actualizeMinlist	**(*)**	connects Minlist to all new open nodes and actualizes distance list
calcMinimum	**(1)**	computes minimum of the current distance list
solution	**(*) =(0)|(1)**	if match: shortest path is found if not: continue with rule expand

finished (no more rule is applicable)

Figure 3: Rule Application Strategy for the Find-Shortest-Path-Example

A comparison of our *Shortest-Path* graph grammar solution with a possible textual implementation of the algorithm given in [Win84] in an imperative programming language yields the following advantages of our visual approach:
The problem which is given as a graph (city/ road map), can be implemented directly and needs not be translated to textual data structures like linked lists, matrices, etc.
The underlying structure of the solution algorithm is a tree, and hence can also be implemented visually, which makes the algorithm easy to understand and to debug, especially as the algorithm steps are realized by rule applications, and their effects are shown directly in the current state graph during interpretation.
The computation of the minimal path length is not done visually in our example but by using the sophisticated Java class `OrderedSet` from the Java library `JGL`. A visual realization would be rather complex and result in several additional rules. These rules should be hidden from the main algorithm rules to obtain a well structured implementation. Up to now, a hierarchy of rules cannot be realized in AGG.
Control Structures like loops are given implicitly by the AGG inherent rule application strategy, i.e. by multiple rule applications and by jumping back to the first grammar rule after all rules have been applied. A loop termination condition is reached if there are no more matches for the current rule. The program terminates if there are no more matches for any of the rules.

References

[Win84] P. H. Winston. *Artificial Intelligence*. Addison Wesley, New York, 1984.