Metro Planner: Algorithms

Version	Data	Comments
0.1	2005-02-15	Tried out during lecture of 2005-02-15

Introduction

This note gives a brief summary of some graph algorithms and their adaptation to the problem of finding an optimal route in a metro network. The following subjects are discussed in a sequence of refinement steps:

- Problem statement;
- Reachability in graph;
- Breadth First Search (BFS);
- Adaptation of BFS to metro network;
- Minimal transfers vs. minimal stops;
- More general solutions.

Essential Problem: Shortest Path in Graph

Problem

Given graph G = (N, E), where:

- N is a set (nodes)
- $E \subseteq N \times N$ (edges)
- $s \in N$ (source)
- $t \in N$ (target)
- (optionally: $w \in N$? Integer, a weight function assigning a weight to each edge)

Find optimal path from s to t , where optimal can be:

- Minimum number of edges;
- Minimum cost (sum of the cost/weight of all edges on the path);
- Some combination of these.

Variations on the problem

- All pairs shortest path;
- Negative weight of edges.

General Solutions

- Dijkstra's algorithm (for non-negative weights of edges);
- Bellman-Ford algorithm (allows negative weights, if not in cycles)

When all edges have equal weight, simpler solutions are possible, based on Breadth First Search.

Reachability in Graph

As a first step, consider a simpler problem:

Given graph G = (N, E), where:

- N is a set (nodes)
- $E \subseteq N \times N$ (edges)
- $s \in N$ (source)

Find the set of nodes reachable from s.

Usually stated in terms of graph coloring:

- Initially, all nodes are colored white;
- Finally, all and only nodes reachable from s are colored black.

Algorithm is based on three-color scheme, with the following intuitive meaning:

- White: undiscovered;
- Grey: discovered (i.e. reachable from s, outgoing edges not yet traced);
- Black: explored (i.e. reachable from s, all outgoing edges traced).

Initially, s is colored grey, all other nodes white; finally, no grey nodes left.

Algorithm:

- W: set of white nodes
- G: set of grey nodes
- B: set of black nodes

```
B, G, W := Ø, {s}, N - {s}   // color s grey, all other nodes white
while G ? Ø do
    let g ∈ G;  //N.B. non-deterministic choice
    forall h such that (g,h)∈ E and h ∈ W  do
        W, G := W - {h}, G ∪{h}   // color all white successors grey
    od;
    G, B := G - {g}, B ∪ {g}      // color  g  black
od
```

Breadth First Search

Refinements and additions:

- Organize G as a FIFO queue Q. Thus, the oldest unexplored nodes are visited first. This leads to a particular visit order known as *breadth-first search* (BFS), where nodes are visited in an order corresponding to their distance from s.
- Construct a breadth first search tree, i.e. a tree where each node (apart from the root) has a parent pointer to the node from which it was discovered. Following these pointers leads to the root s in a minimal number of steps.

Also:

- Do bookkeeping locally in each node: color, parent, distance;
- Optionally, provide "hooks" for processing a node or an edge.

This leads to the following algorithm:

```
forall g in N - {s} do g.color := white; g.parent := nil; g.dist := +∞ od;
s.color := grey; s.parent := nil; s.dist := 0;
Q := <s>;
while Q ? <> do
    g, Q := first(Q), rest(Q);
    "process g";
    forall h such that (g,h) in E and h.color = white do
        h.color := grey; Q := Q ++ <h>; h.parent := g; h.dist := g.dist = 1;
        "process (g,h)";
    od;
    g.color := black
od;
```

If the goal is just to determine a shortest path from the source `s` to a given target `t`, the search can stop as soon as the color of `t` changes from white to grey (if at all). This can be achieved by strengthening the guard of the while loop with the conjunct `t.color ? grey`. The path from `s` to `t` can be constructed by following the parent path from `t` to `s`. Applying these changes leads to the following algorithm:

```
{initialization}
forall g in N - {s} do g.color := white; g.parent := nil; g.dist := +∞ od;
s.color := grey; s.parent := nil; s.dist := 0;
Q := <s>;
{main loop}
while (Q ? <>) and (t.color ? grey) do
    g, Q := first(Q), rest(Q);
    "process g";
    forall h such that (g,h) in E and h.color = white do
        h.color := grey; Q := Q ++ <h>; h.parent := g; h.dist := g.dist = 1;
        "process (g,h)";
    od;
    g.color := black
od;
{(Q = <>) or (t.color = grey)}
if t.color ? grey
then {t is not reachable from s} path := <>
else {construct path by following parent pointers}
```

```
        p := t; path := <t>;
        while p ? nil do
            path := <p.parent> ++ path;
            p := p.parent
        od
    fi
    {path is a shortest path from s to t}
```

Adaptation to Metro Network

In the Metro Planner a network is not just a graph. A network is given by:

- SS (a set of stations);
- LS (a set of lines; each line is a sequence of elements from SS, possibly with some additional conditions (no duplicates, one-way, circular, ...).

This structure leads to a particular way of determining the successors of a station (node) U. For each line L, if U occurs on L at some index position I , then its successors are at positions (I+1) and (I-1), roughly speaking. If I is the last position of L , then there is no successor at position (I+1), but if the line is circular there is a successor is at position 0. A similar situation occurs at the first position of L. Moreover, if the line is one-way, then the position (I-1) should not be taken into account.

Another difference concerns the reconstruction of the path. In the Metro Planner, a path is not just a sequence of stations, but the line should be recorded as well. This can be achieved by registering in each station not only the parent, but also the line involved. The path can then be constructed as an alternating sequence of stations and lines.

All in all, this leads to the following adapted version of the algorithm:

```
{initialization}
forall U in SS - {S} do
    U.color := white; U.parent := nil; U.line := nil; U.dist := +∞
od;
S.color := grey; S.parent := nil; S.line := nil; S.dist := 0;
Q := <S>;
{main loop}
while (Q ? <>) and (T.color ? grey) do
    U, Q := first(Q), rest(Q);
```

```
        "process U";
        forall L in LL do
            if U occurs on L at some position I then
                for possible successors V of U on L do
                    V.color := grey; V.parent := U; V.line := L;
                    V.dist := U.dist + 1; Q := Q ++ <V>;
                od;
            fi;
            "process (U,V)";
        od;
        U.color := black
    od;
    {(Q = <>) or (T.color = grey)}
    if T.color ? grey
    then {T is not reachable from S} path := <>
    else {construct path by following parent pointers}
        P := T; path := <T>;
        while P ? nil do
            path := <P.parent,P.line> ++ path;
            P := P.parent
        od
    fi
    {path is a shortest path from S to T}
```

Minimal Transfers vs. Minimal Stops

For a metro network, a minimal number of stops is just one possible notion of optimality. Another possibility is to minimize the number of transfers. One way of doing this is to apply the algorithm above to a reduced network, consisting of just the transfer stations and of the source s and the target t. This solution is not elaborated in this document.

A different solution is the application of Breadth First Search to *lines* rather than *stations*. Consider the start station S and the lines on which it occurs. All stations on these lines can be reached with 0 transfers. Next, from the undiscovered ("white") lines, consider those that intersect with the ("grey") lines considered thus far. All stations on these lines can be reached from S with one transfer. Continuing this way, we perform essentially a breadth first search on lines until a line containing the target station T is reached. This leads to the following algorithm (note that in this case bookkeeping is done in lines, not in stations):

```
{initialization}
Q := <>; targetline := nil;
forall L in LS do
    L.parent := nil;
    if S occurs on L
    then L.color := grey; L.entry := S; Q := Q ++ <L>;
         if T occurs on L then targetline := L fi
    fi
od;
{main loop}
while (Q ? <>) and (targetline = nil) do
    L, Q := first(Q), rest(Q);
    forall stations U occurring on L do
        forall lines H in LS do
            if (U occurs on H) and (H.color = white)
            then H.parent := L; H.entry := U; H.color := grey; Q := Q ++ <H>;
                 if T occurs on H then targetline := H fi
            fi
        od
    od;
    L.color := black
od;
{ (Q = <>) or (targetline = nil) }
if targetline = nil
then {T is not reachable from S} path := <>
else {construct path by following parent pointers}
    L := targetline; path := <T>;
    while L ? nil do
        path := <L.entry, L> ++ path;
        L := L.parent
    od
fi
{path is a shortest path from S to T}
```

More General Solutions

The solutions presented above either minimize the number of stops or the number of transfers. It would be better, however, to optimize some (adjustable) weighted combination of both. This leads to a more complicated cost function and requires a more general shortest path algorithm like the one by Dijkstra or Bellman-Ford. These solutions will not be elaborated here.