# Dinc's Algorithm for Maximum Flows

Jonathan Turner

April 2, 2013

When we first studied the maximum flow problem in graphs, we described the augmenting path method for finding maximum flows. When augmenting paths are selected in increasing order of their length, this yields an $O(m^2n)$ time algorithm. In this section, we'll consider a refinement of the shortest augmenting path algorithm that runs in $O(mn^2)$ time, making it faster by a factor of $m/n$, a substantial advantage for dense graphs.

Recall that when analyzing the performance of the shortest augmenting path algorithm, we defined $R_i$ to be the residual graph after the $i$-th augmenting path step and $level_i(u)$ to be the number of edges in a shortest path in $R_i$ from the source to $u$. We also showed that $level_i(t)$ is a non-decreasing function of time and that it cannot remain unchanged for more than $m$ augmenting steps.

We can view the execution of the augmenting path algorithm as being divided into a series of phases of at most $m$ steps each, where a phase ends whenever $level_i(t)$ increases. If $R_i$ is the residual graph at the start of some phase, then during that phase, all augmenting paths are selected from among those edges $(u, v)$ in $R_i$ with $level_i(v) = level_i(u) + 1$. We can refer to these as the *eligible edges* for the phase. Dinic's algorithm makes this division into phases explicit and during a phase it restricts its attention to these eligible edges.

Unlike the shortest augmenting path algorithm, which uses a breadth-first search, Dinic's algorithm uses a depth-first search in the subgraph defined by the eligible edges. Because this subgraph is acyclic, each edge explored by the algorithm either leads to the sink or to a dead-end. The algorithm is designed to avoid dead-ends once they are first discovered, allowing it to get substantially better running time than the shortest augmenting path algorithm.

There are two approaches to implementing Dinic's algorithm. The first involves explicitly constructing the subgraph defined by the eligible edges at the start of each phase. Then, during each augmenting path search within a phase, we do a depth-first search of the subgraph. Any edges that lead to dead-ends are removed from the subgraph so that later searches won't waste time exploring them again.

The second approach skips the construction of the subgraph, using another method to avoid repeated examination of dead-ends. A the start of each phase, the algorithm computes $\mathit{level}(u)$ for each vertex $u$ using a breadth-first search. During each augmenting path search, only edges $(u, v)$ in the residual graph with $\mathit{level}(v) = \mathit{level}(u) + 1$ are considered. All others are simply skipped.

The algorithm also uses a pointer $\mathit{nextedge}(u)$ which is initialized to the first edge in $u$'s adjacency list. The *nextedge* pointers are advanced along the adjacency list anytime an edge is examined without resulting in an augmenting path. This is sufficient to avoid repeated examination of deadends. Figure 1 illustrates the use of *nextedge*.

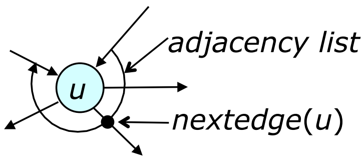

Figure 1: Nextedge pointer in augmenting path searches

With this context, we can write the *findpath* method in algorithmic notation as follows.

```
predicate findpath(vertex $u$)
    do $\mathit{nextedge}(u) \neq$ null $\Rightarrow$
        $(u, v) := \mathit{nextedge}(u)$;
        if $\mathit{res}(u, v) > 0$ and $\mathit{level}(v) = \mathit{level}(u) + 1$ and
            $(v = t$ or $\mathit{findpath}(v)) \Rightarrow$
            $\mathit{pEdge}(v) := (u, v)$;
            return true;
        fi;
        advance $\mathit{nextedge}(u)$;
    rof;
    return false;
end;
```

If we invoke this method from the source vertex, it will return true if and only if there is an augmenting path using the eligible edges. It will also update the *pEdge* values to identify the augmenting path found, and it will *advance* the *nextedge* pointers past any edges that lead to dead-ends. This is how it avoids repeated exploration of those edges during later searches.