Olympiads in Informatics, 2018, Vol. 12, 25–41
© 2018 *IOI, Vilnius University*
DOI: 10.15388/ioi.2018.03

Tidal Flow: A Fast and Teachable Maximum Flow Algorithm

Matthew C. FONTAINE
Independent Researcher
e-mail: tehqin@gmail.com

Abstract. Among the most interesting problems in competitive programming involve maximum flows. However, efficient algorithms for solving these problems are often difficult for students to understand at an intuitive level. One reason for this difficulty may be a lack of suitable metaphors relating these algorithms to concepts that the students already understand. This paper introduces a novel maximum flow algorithm, Tidal Flow, that is designed to be intuitive to undergraduate and pre-university computer science students.

Keywords: algorithms, maximum flow, flow network, flow augmentation.

1. Introduction

Maximum flows are a well-researched area in optimization theory. The problem was originally formulated by Harris and Ross (1955) and solved by using the well-known augmenting path technique by Ford and Fulkerson (1955). Since this initial discovery, many maximum flow algorithms have been developed (for a survey see: Ahuja *et al.*, 1993; Goldberg and Tarjan, 2014). Most of these algorithms are space efficient. As a result, time complexity becomes the primary basis of comparison between maximum flow algorithms (Goldberg and Tarjan, 2014). The maximum flow formulation opens itself to a wide variety of applications (for examples see: Ahuja *et al.*, 1993). This diversity of applications has increased the popularity of problems involving maximum flows in the competitive programming community. Problems that require the creation of networks with large capacities and a large number of vertices and edges demand the use of faster flow algorithms to solve each problem in a reasonable amount of time. These more complex algorithms can be a burden for students to understand. As proposed by Forišek and Stienová (2013), there are differences among deriving, proving, and teaching algorithms. How easy an algorithm is to teach is a factor in how widely adopted an algorithm will be. Moreover, the theoretically best algorithms with respect to time complexity are not always the fastest when implemented in practice (Ahuja *et al.*, 1997), i.e. in a programming contest setting. These considerations create a need for a fast flow algorithm that is easy to teach, is easy to implement, and offers competitive in-practice performance compared to other popular flow algorithms. This paper proposes Tidal Flow as an algorithm to satisfy these three objectives.

2. Background

This section reviews the maximum flow problem and introduces the notation that will be used throughout the rest of the paper. It then describes three historically important maximum flow algorithms that introduce necessary concepts for understanding the novel Tidal Flow algorithm.

2.1. *Maximum Flow Formulation and Notation.*

This paper will use formulation and notation adapted from Goldberg and Tarjan (2014). The input to the maximum flow problem is (G, s, t, cap), where $G = (V, E)$ is a directed graph with n vertices and m edges. The input marks two special vertices s, $t \in V$. The vertex s is known as a source and the vertex t is known as the sink. The function $cap : E \rightarrow \mathbb{R}^+$ is some strictly positive capacity function. A maximum flow is some non-negative function $f : E \rightarrow \mathbb{R}^*$ that satisfies two constraints: (1) a capacity constraint $f(e) \leq cap(e)$, $\forall e \in E$ and (2) a conservation constraint $\sum_{(w,v)\in E} f(w,v) = \sum_{(v,u)\in E} f(v,u)$, $\forall v \in V - \{s,t\}$. The capacity constraint ensures that flow sent down some edge e does not exceed its capacity, while the conservation constraint maintains the flow entering some vertex v equals the amount of flow leaving v. The conservation constraint is maintained for all vertices except the source and sink. The flow value is defined as the amount of flow leaving the source or $\sum_{v\in V} f(s,v)$. A maximum flow is one that maximizes the flow value subject to the conservation and capacity constraints.

2.2. *Residual Graphs and Augmenting Paths.*

To make it easier to discover maximum flows, it is useful to make the graph more malleable. Residual graphs are a useful tool for this purpose. Consider each edge $(w,v) \in E$. To change the flow along this edge, one could increase the flow by up to $cap(w,v) - f(w,v)$ or decrease the flow by $f(w,v)$. Decreasing the flow is easier to manage by including a reverse edge (v,w) with $cap(w,v)$ where $f(v,w) = -f(w,v)$ at all times. Now, decreasing flow $f(w,v)$ can be accomplished by increasing $f(v,w)$. The amount that $f(w,v)$ can be decreased is given by $cap(v,w) - f(v,w)$. Now consider a new edge set E' that contains all edges of E and all reverse edges of E. The residual graph is $G_r = (V, E')$ with a new function $cap_r(w,v) = cap(w, v) - f(w,v)$. Conceptually, cap_r gives a limit on how much each $e \in E'$ can change along that direction.

To modify f and maintain conservation and capacity constraints, an algorithm can discover some path P from source to sink where each edge on that path e has $cap_r(w, v) > 0$. The algorithm can then augment each edge $e \in P$ by $a = min_{e \in P}(cap_r(e))$.

This change can be accomplished through modifying f: $f(w, v) \longleftarrow f(w, v) + a$ and $f(v, w) \longleftarrow f(v, w) - a$ for each edge $(w, v) \in P$. P is known as an *augmenting path* (Ford and Fulkerson, 1955). Each augmentation maintains the capacity and conservation constraints and gradually increases the flow with each augmentation until a maximum flow is reached. Augmenting paths can be applied between any two vertices $w,\ v \in V$. The term *global augmenting path* refers to augmenting paths between s and t to distinguish this approach from techniques that make local improvements to flow.

2.3. *Dinitz's Algorithm and Level Graphs.*

Augmenting along shortest paths was discovered to be more efficient than augmenting along other types of paths (Dinitz, 1970; Edmonds and Karp, 1972). Dinitz's algorithm works by efficiently computing a level graph from s to t and finding paths from s to t through that level graph.

A *level graph* $G_L = (V_L, E_L)$ is a graph where V_L contains all vertices $v \in V$ on all shortest paths from s to t in G (see Fig. 1). Let $d : V \rightarrow \mathbb{Z}^*$denote the number of edges on a path from s to $v \in V$ with minimum number of edges. E_L contains all edges in $(w, v) \in E$ where $cap_r(w, v) > 0$ and $d(w) + 1 = d(v)$.

Dinitz's algorithm computes a level graph using a breadth first search (BFS). The algorithm then attempts to saturate enough edges in G_L to prevent any augmenting path from s to t. An edge $e \in E$ is *saturated* if $cap_r(e) = 0$. Such a flow is called a *blocking flow* of level graph G_L.

Shimon Even revised and popularized Dinitz's algorithm, creating the well-known version (Dinitz, 2006). (When popularizing the algorithm, Even spelled Dinitz as Dinic and changed the pronunciation (Dinitz, 2006).) In Even's version, a blocking flow is computed through a modified depth first search (DFS). The DFS finds a maximal set of shortest augmenting paths, which is sufficient to block G_L. The modified DFS produces these augmenting paths in $O(nm)$ running time. The BFS and DFS procedures repeat until no augmenting path exists from source to sink. Each level graph blocked increases the length of the shortest path in the next discovered level graph, resulting in an $O(n^2m)$ total running time. The original algorithm also computes a blocking flow of this level graph in $O(nm)$, but is both more complicated conceptually and harder to implement.

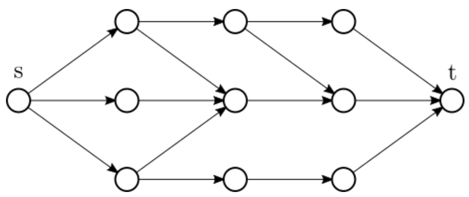


Fig. 1. A level graph.

2.4. *Karzanov's Algorithm and Preflows.*

Karzanov formalized the concept of blocking flows and additionally introduced the concept of preflows (Karzanov, 1974). In Karzanov's algorithm, preflows are used to in place of global augmenting paths to block the level graph. A *preflow* is similar to a flow f, but the conservation constraint is removed, meaning more flow can go into a vertex than is leaving it. To help keep track of the amount of extra flow at some vertex a function $ex : V \to \mathbb{R}^*$ is used, where $ex(v) = \sum_{(w,v)\in E} f(w,v) - \sum_{(v,u)\in E} f(v,u)$.

Similar to Dinitz's algorithm, Karzanov's algorithm repeatedly computes level graphs and blocks them. For each level graph, the algorithm sends preflows through the level graph and gradually restores the conservation constraint, converting the blocking preflow into a blocking flow. The blocking flows are discovered in $O(n^2)$ running time, resulting in a $O(n^3)$ running time for the algorithm.

3. Tidal Flow

This section introduces the Tidal Flow algorithm by first explaining its relationship to Dinitz's and Karzanov's algorithms, then introducing an ocean tide metaphor that makes the algorithm easier to understand at a high level, and then finally explaining the technical details and formalization of the algorithm.

3.1. *Blocking Flows.*

Like Dinitz's algorithm and Karzanov's algorithm, Tidal Flow computes blocking flows on a level graph. Like Dinitz's algorithm, Tidal Flow's goal is to produce blocking flows by discovering global augmenting paths. However, instead of computing the blocking flow in $O(nm)$, the Tidal Flow attempts to compute a blocking flow in $O(m)$. Unlike Dinitz's algorithm, Tidal Flow makes no guarantee to discover a blocking flow in the level graph in one pass. To simplify the blocking procedure, the edges of the level graph are stored as a list in BFS order (Fig. 2).

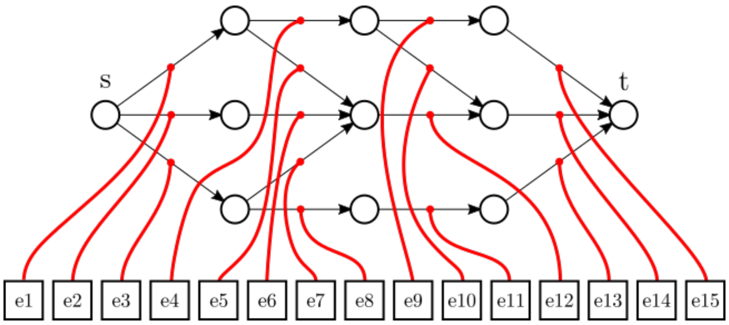


Fig. 2. A level graph stored as a list of edges.

3.2. *Tide Metaphor.*

Metaphors can be a powerful tool in teaching new algorithms. Forišek and Stienová give the following definition:

> A (conceptual) metaphor is a cognitive process that occurs when a subject seeks understanding of one idea (the target domain) in terms of a different, already known idea (the source domain). The subject creates a conceptual mapping between the properties of the source and the target, thereby gaining new understanding about the target. (Forišek and Stienová, 2013, adapted from Lakoff and Johnson, 2003)

Tidal Flow uses a conceptual metaphor based on oceanic tide cycles to help explain its level graph blocking procedure.

3.3. *Discovering Blocking Flows as Tides.*

The goal of Tidal Flow is to produce a blocking flow on a level graph. Tidal Flow does that through a procedure called *tide cycle*.

Tide cycle has three phases: high tide, low tide, and erosion.

(1) **High tide:** Produce an upper bound on the amount of flow that can reach each vertex in the level graph by passing from source to sink.
(2) **Low tide:** Reduce the amount of flow that can reach each vertex to a feasible amount by passing from sink to source.
(3) **Erosion:** Change the flow on each edge used and update residual flow.

In terms of the metaphor, vertices are *tide pools* that temporarily collect flow during each phase of tide cycle. Tide pools store an upper bound on the amount of flow that can reach each vertex during high tide when flow passes from source to sink. During low tide, flow is pushed back from sink to source. Not all flow will make it back to the source and some tide pools retain excess flow (similar to how tide pools in nature exist as separate bodies of water during low tide). Because the level graph is stored as a list of edges, each phase can be implemented as a loop through the list of edges.

Consider the example level graph in Fig. 3. During the high tide phase, a function $h : V \to \mathbb{R}^*$ is computed storing an upper bound on the amount of flow that can reach each vertex. In terms of the metaphor, $h(v)$ is the amount of flow stored in

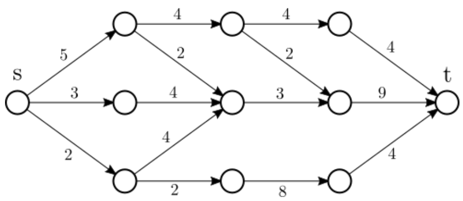


Fig. 3. Example level graph with edge capacities.

each tide pool at high tide. $h(v)$ can be any upper bound on flow that can reach vertex v through a set of augmenting paths that collectively maintain the capacity constraint. One could consider h to be a heuristic guessing function for Tidal Flow, similar to the heuristic function in the A* shortest path algorithm (Lerner *et al.*, 2009). For simplicity, $h(v)$ is computed as the sum of $h(w)$ for each edge (w, v) that enters v bounded by that edge's capacity (Equation 1, Fig. 4). Flow is promised to each edge $(w, v) \in E$ of the level graph $p(w, v) = \min(cap(w, v) - f(w, v), h(w))$.

$$h(v) = \sum\nolimits_{w \in G(v)} \min(cap(w, v) - f(w, v), h(w)) \tag{1}$$

During the low tide phase, flow is pushed from sink to source backwards through the level graph using h and p as guides for how much flow to push to each vertex. A new function $l : V \rightarrow \mathbb{R}^*$ maintains the amount of flow in tide pool v. $l(t)$ is initialized with $h(t)$. When an edge (w, v) is evaluated on the way back to the source, flow is drained from $l(v)$ and transferred to $l(w)$ and the promised flow $p(w, v)$ is updated by this transferred flow (Fig. 5).

In the erosion phase, the promised flow is committed to the network. In the example, 11 units of flow were promised from source to sink, but only 9 units of flow are committed. In the example, Tidal Flow manages to find a blocking flow in one pass of tide cycle.

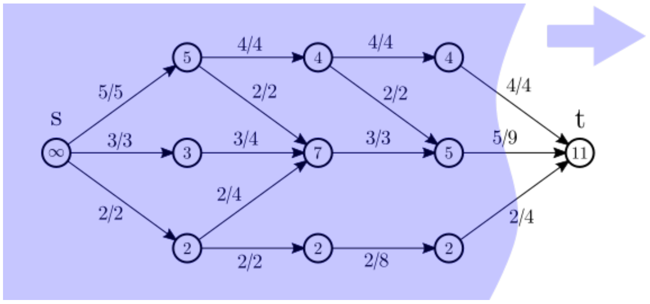


Fig. 4. High tide calculates $h(v)$ for each vertex v. Edges store promised flow $p(e)/cap_r(e)$.

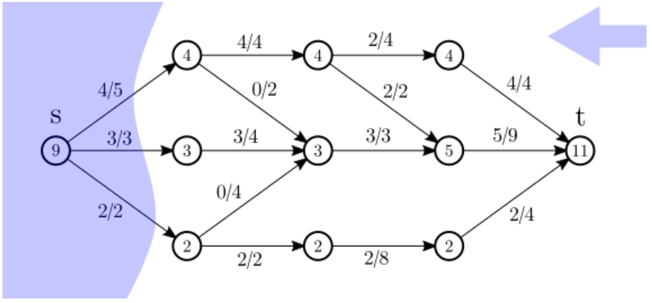


Fig. 5. Low tide pushes flow from t to s. Here each vertex contains $l(v)$ at its highest point during low tide.

3.4. *Relationship to Preflows.*

The low tide phase of tide cycle is a type of preflow. Excesses are placed at the sink and as much flow as possible is pushed between nodes. The preflow values never augment the network, so Tidal Flow always maintains the conservation constraint. Instead, preflows are used to discover a set of potentially overlapping augmenting paths that collectively maintain the capacity constraint and augment across those paths in parallel. In this sense Tidal Flow is both a preflow algorithm and an augmenting path algorithm.

Algorithm 1: Attempt to compute a blocking flow

```
TideCycle ($E$)
    input : A list $E$ of level graph edges in BFS order.
    output: The amount of flow sent through the level graph.
    result : $f$ is modified by found augmenting paths.
    $h(v) = 0, \forall v \in V$;
    $h(source) \longleftarrow \infty$;
    foreach edge $e_i(w, v) \in E$ do
        $p(e_i) \longleftarrow \min(cap(e_i) - f(e_i),\ h(w))$;
        $h(v) \longleftarrow h(v) + p(e_i)$;
    end
    if $h(sink) = 0$ then
        return 0;
    end
    $l(v) = 0, \forall v \in V$;
    $l(sink) \longleftarrow h(sink)$;
    foreach edge $e_i(w, v) \in E$ in reverse order do
        $p(e_i) \longleftarrow \min(p(e_i),\ h(w) - l(w),\ l(v))$;
        $l(v) \longleftarrow l(v) - p(e_i)$;
        $l(w) \longleftarrow l(w) + p(e_i)$;
    end
    $h(v) = 0, \forall v \in V$;
    $h(source) \longleftarrow l(source)$;
    foreach edge $e_i(w, v) \in E$ do
        $p(e_i) \longleftarrow \min(p(e_i),\ h(w))$;
        $h(w) \longleftarrow h(w) - p(e_i)$;
        $h(v) \longleftarrow h(v) + p(e_i)$;
        $f(e_i) \longleftarrow f(e_i) + p(e_i)$;
        $f(rev(e_i)) \longleftarrow f(rev(e_i)) - p(e_i)$;
    end
    return $h(sink)$;
```

4. Evaluating the Performance of Tidal Flow

4.1. *Correctness.*

Flow is only modified through augmenting paths, which maintain the capacity and conservation constraints. During each tide cycle, the amount of flow increases until no augmenting paths exist. Using the usual arguments involving augmenting paths, Tidal Flow will terminate with a maximum flow.

4.2. *Theoretical Performance of Tidal Flow.*

Edmonds and Karp (1972) introduced an argument for bounding the running time of the shortest augmenting path method of finding maximum flows. The tide cycle procedure in Tidal Flow will always fully saturate at least one edge on a shortest augmenting path. This gives an upper bound on the time complexity of Tidal Flow to be at most $O(nm^2)$. This bound may not be tight. Tide cycle will regularly find several augmenting paths.

4.3. *Difficulty of Bounding Tide Cycles.*

Determining the upper bound on the number of tide cycles required to produce a blocking flow is difficult. During high tide, a network can trick the heuristic function h into promising much more flow than is feasible to realize during low tide by creating a lens (Fig. 6) somewhere in the network. A *lens* is a dense level subgraph with two node levels where edges double in capacity each level.

Lenses require $\approx log(k)$ nodes to magnify the flow through the network by k. Using lenses, it is possible to construct level graphs requiring $O(n/log(n))$ tide cycles to block. Fig. 7 provides one such construction. Creating level graphs requiring more tide cycle operations is not obvious.

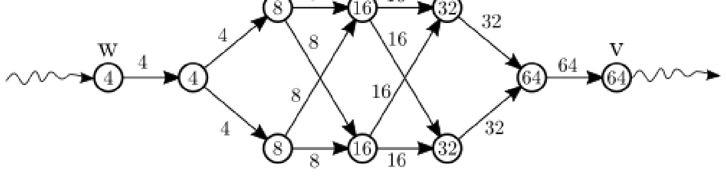


Fig. 6. A lens. Only 4 units of flow can pass from w to v, yet 64 units are promised.

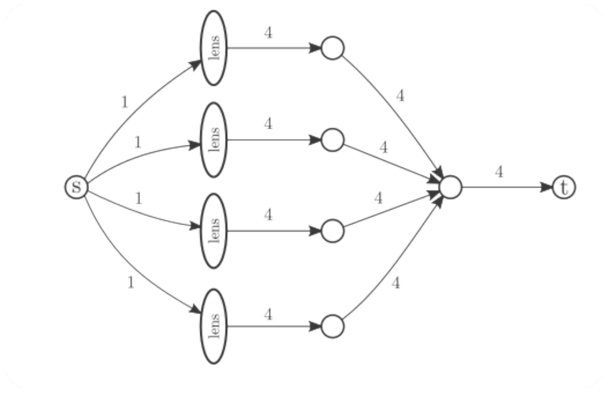


Fig. 7. A level graph that takes 4 tide cycles to block. The graph can be generalized to a graph with n vertices requiring $O(n/log(n))$ tide cycles.

4.4. *Evaluating Practical Performance.*

Many of the theoretically fastest maximum flow algorithms are much slower in practice (Goldberg and Tarjan, 2014). The fastest theoretical algorithm for maximum flows is due to Orlin (2013) and achieves a $O(nm)$ running time. The overhead of the algorithm and rarity of the worst cases causes the best theoretical approach to be defeated in practice by slower algorithms (Boykov and Kolmogorov, 2004; Goldberg *et al.*, 2011). Though many flow algorithms have large upper bounds on running time, they rarely achieve this behavior in the average case.

To evaluate Tidal Flow, the algorithm was benchmarked against several maximum flow algorithms (Table 1) that are known to do well in practice (Ahuja *et al.*, 1997)

Table 1
Flow algorithms

Algorithm	Running Time	Notes
Edmonds-Karp (Edmonds and Karp, 1972)	$O(nm^2)$	Shortest augmenting path
Dinitz (Dinitz, 1970)	$O(n^2m)$	Even's version with optimizations suggested in Dinitz (2006)
Preflow-Push (Goldberg and Tarjan, 1988)	$O(n^3)$	Goldberg and Tarjan's preflow-push algorithm with the highest-label selection rule. A simple, but inefficient, selection implementation process yields the slower runtime
Preflow-Push (Gap) (Goldberg and Tarjan, 1988)	$O(n^2\sqrt{m})$	Goldberg and Tarjan's push relabel method with the highest-label selection rule. Implemented with $O(1)$ selection and the gap relabeling heuristic suggested in Cherkassy and Goldberg (1995)
Improved-SAP (Orlin and Ahuja, 1987)	$O(n^2m)$	Orlin's improved shortest augmenting path method

and are popular in competitive programming. Each flow algorithm was run against a test suite of randomly generated networks that resemble classes of graphs common in competitive programming network flow problems. The algorithms were implemented in Java and each experiment measured CPU time using the StopwatchCPU class from Sedgewick and Wayne (2011).

Three forms of graphs were tested: bipartite matching networks, grid networks, and level graphs (Fig. 8). Bipartite matching networks were split into four different graph classes, resulting in six total graph classes. Each graph class was evaluated at 10 different sizes (described below). For each size of graph in each graph class, 20 random graphs were generated. Each algorithm was run against each test for a maximum of 20 seconds.

(1) **Dense-highcap-bpm:** A fully connected bipartite matching graph. Capacities for internal edges were selected uniformly at random from $[1, 1000]$. Capacities for edges (s, v) and (v, t) were selected from $[1, c(v)]$ where $c(v) = \sum_{(v,w)\in E} cap(v, w)$. Graph size n represents the number of internal vertices. A range of sizes $n = [200, 2000]$ were selected in increments of 200.

(2) **Sparse-highcap-bpm**: A bipartite matching graph. Each vertex v on the source side of the bipartite graph was connected with $\sqrt{n}$ random neighbors on the sink side. Capacities for internal edges were selected uniformly at random from $[1, 10000]$. Capacities for edges (s, v) and (v, t) were selected from $[1, c(v)]$ where $c(v) = 1 + \sum_{(v,w)\in E} cap(v, w)$. Graph size n represents the number of internal vertices. A range of sizes $n = [1000, 10000]$ were selected in increments of 1000.

(3) **Dense-unit-bpm:** A bipartite matching graph where all edge capacities are unit capacities. Each vertex v on the source side of the bipartite graph was connected with 10 random neighbors on the sink side. Graph size n represents the number

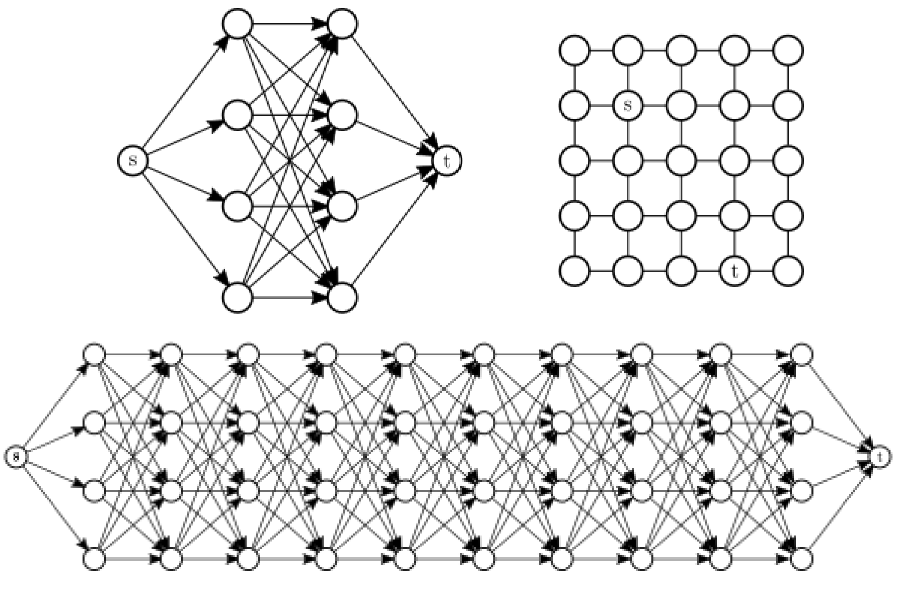


Fig. 8. Three forms of graph classes.

of internal vertices. A range of sizes $n = [28000, 10^5]$ were selected in increments of 8000.

(4) **Sparse-unit-bpm:** A bipartite matching graph where all edge capacities are unit capacities. Each vertex v on the source side of the bipartite graph was connected with 10 random neighbors on the sink side. Graph size n represents the number of internal nodes. A range of sizes $n = [28000, 10^5]$ were selected in increments of 8000.

(5) **Grid:** An $n \times n$ grid network where each neighbor in the four cardinal directions is connected. The source and sink were selected uniformly at random among grid vertices. Edge capacities were selected uniformly at random from $[1, 10^8]$. Graph size n represents an $n \times n$ grid of vertices. A range of sizes $n = [275, 500]$ were selected in increments of 25.

(6) **Level-10:** A level graph with 10 levels where level i is fully connected to level $i+1$. Capacities for internal edges were chosen uniformly at random from $[1, 1000]$. Capacities for edges (s, v) and (v, t) were selected from $[1, c(v)]$ where $c(v) = \sum_{(v,w) \in E} cap(v, w)$. Graph size n represents a $10 \times n$ level graph. A range of sizes $n = [140, 500]$ were selected in increments of 40.

5. Results

Fig. 9–Fig. 14 compare Tidal Flow against the flow algorithms from Table 1. Flow algorithms that didn't complete a majority of the tests were removed from the figures to make it easier to directly compare Tidal Flow against more competitive algorithms.

Algorithm performance on dense-highcap-bpm (Fig. 9). On this graph class Edmonds-Karp only completes size 200 graphs before timing out on all remaining test sizes. Preflow-Push (Gap) manages to complete up to 1200 size graphs but times out at 1400. The variance of Preflow-Push (Gap) is much higher than other algorithms. Preflow-Push completes graphs up to size 1400, but also has a high variance. ISAP completes graphs up to size 1400. Dinitz manages to complete all tests but performs worse

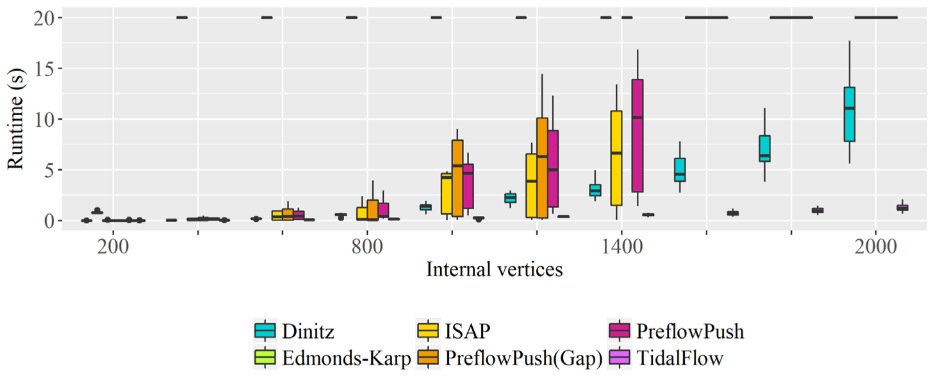


Fig. 9. Algorithm performance on dense-highcap-bpm.

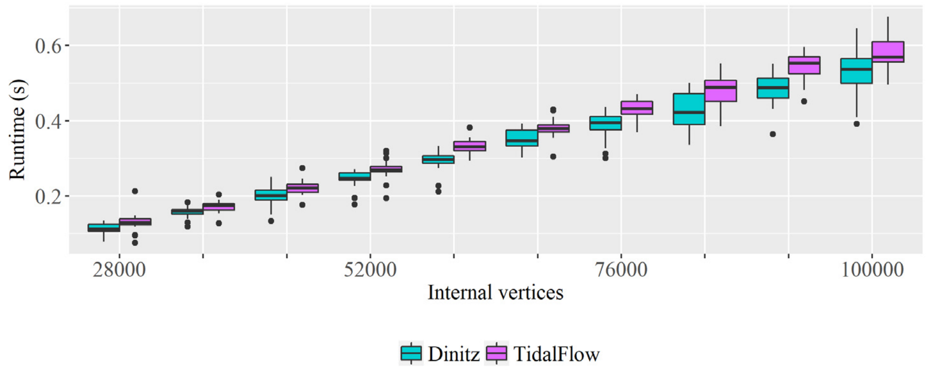


Fig. 10. Algorithm performance on sparse-highcap-bpm.

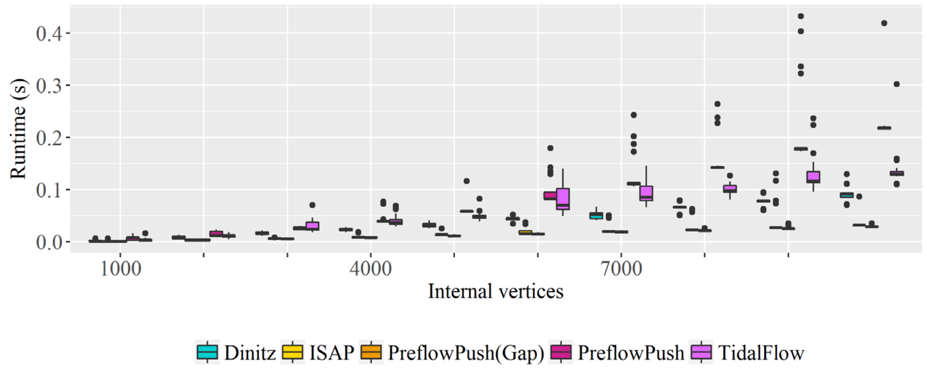


Fig. 11. Algorithm performance on dense-unit-bpm.

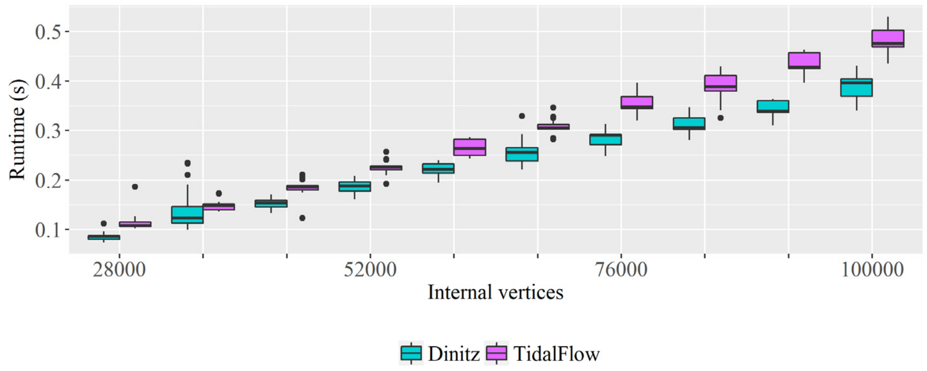


Fig. 12. Algorithm performance on sparse-unit-bpm.

than Tidal Flow. Tidal Flow completes all tests and has much lower variance than any of the other included algorithms.

Algorithm performance on sparse-highcap-bpm (Fig. 10). On this graph only Tidal Flow and Dinitz were able to complete all tests in 20 seconds. Preflow-Push(Gap) was the only other algorithm able to complete a test and only completed the smallest test size in under 12 seconds before timing out. Only Dinitz and Tidal Flow are included in this graph to allow a closer comparison of the two algorithms. Dinitz performs slightly better than on this graph class, but Tidal Flow is comparable in performance. All tests run under 0.7 seconds.

Algorithm performance on dense-unit-bpm (Fig. 11). Dinitz completes all tests in under 0.2 seconds. Edmonds-Karp completes half of the test suite before timing out. ISAP runs all tests in under 0.04 seconds. PreflowPush (Gap) runs all tests in under 0.03 seconds. Preflow-Push runs all tests in under 0.3 seconds. Tidal Flow is comparable to Dinitz, solving all tests in under 0.2 seconds.

Algorithm performance on sparse-unit-bpm (Fig. 12). Dinitz and Tidal flow were the only algorithms that could complete any of the tests. Dinitz performs slightly better than Tidal Flow on this entire test suite.

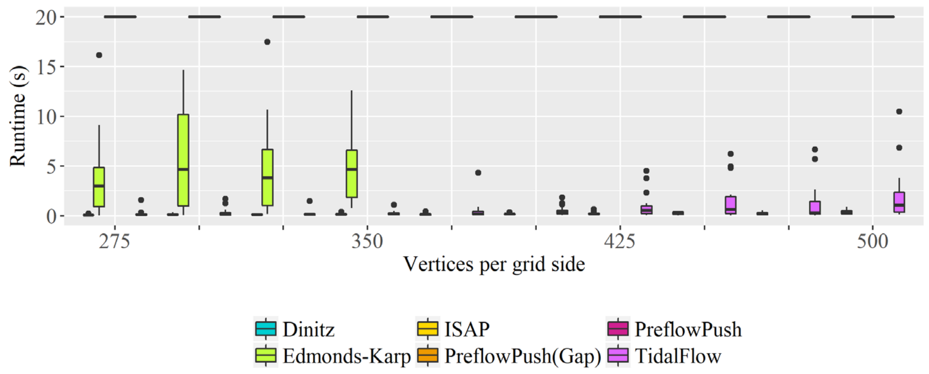


Fig. 13. Algorithm performance on grid.

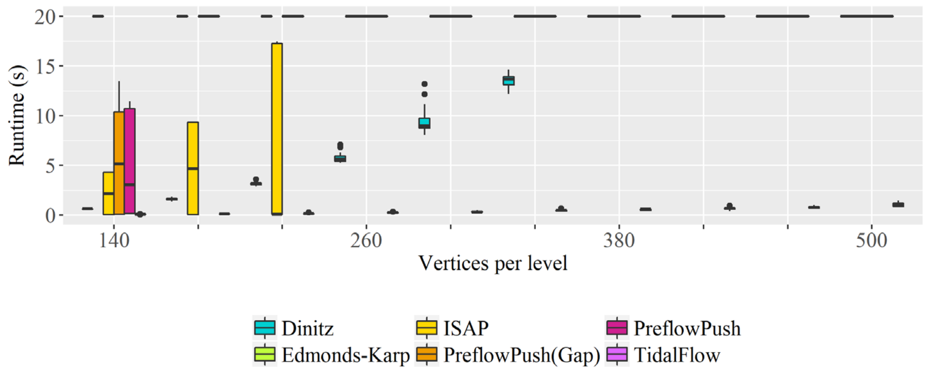
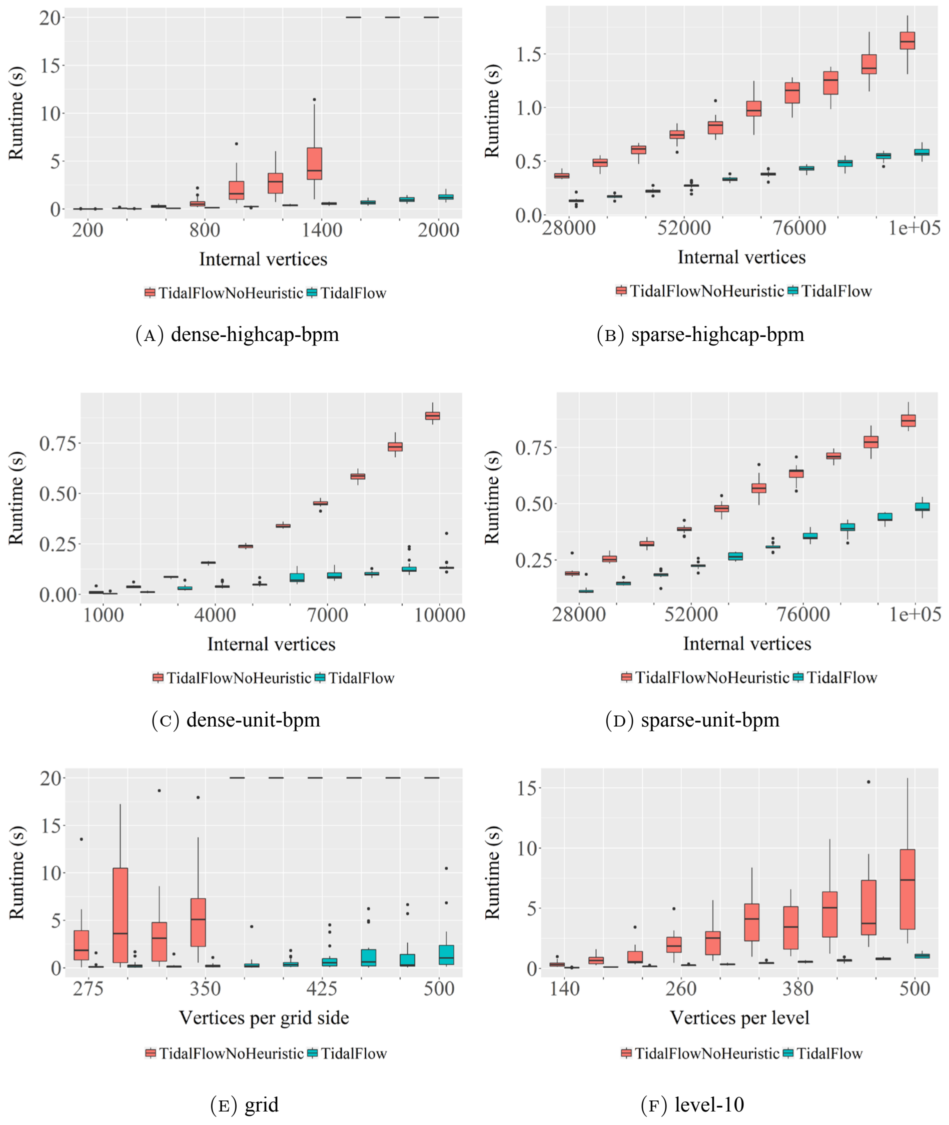


Fig. 14. Algorithm performance on level-10.

Algorithm performance on grid (Fig. 13). Dinitz manages to complete all tests in under 1 second. Edmonds-Karp completes up to size 350 before timing out. TidalFlow's average case is comparable to Dinitz but has several cases much slower.

Algorithm performance on level-10 (Fig. 14). Edmonds-Karp fails to complete any graphs. Dinitz completes graphs up to size 340 before timing out. ISAP can complete graphs up to size 220. PreflowPush(Gap) completes only graphs size 140. Preflow-Push also completes only graphs of size 140. Tidal Flow completes all graphs in less than 2 seconds.



(A) dense-highcap-bpm

(B) sparse-highcap-bpm

(C) dense-unit-bpm

(D) sparse-unit-bpm

(E) grid

(F) level-10

Fig. 15. Running Tidal Flow without the heuristic function h.

In Fig. 15 Tidal Flow using the heuristic function h is compared against a different implementation of Tidal Flow that removes the heuristic function. For every class of graphs, removing the heuristic function resulted in a significantly slower algorithm. The algorithm also started to have a wider variance in running time.

6. Discussion

6.1. *Comparison to other Fast Flow Algorithms.*

Tidal Flow performed well compared to other flow algorithms. Perhaps the most surprising performance was the behavior of Preflow-Push algorithms on the generated graphs. In Ahuja's study (Ahuja *et al.*, 1997) Preflow-Push algorithms perform far better than other algorithms. This is most likely due to Preflow-Push being implemented without the global relabeling heuristic. That heuristic seems to make a huge difference in the behavior of Push-Relabel. In this experiment Dinitz's algorithm significantly outperformed ISAP. In contrast, Ahuja's study found the two algorithms to be comparable. The improvements in Dinitz's algorithm's performance are likely due to the implementation improvements suggested in Dinitz (2006).

Dinitz's algorithm performed better than Tidal Flow on unit capacity bipartite matching cases. Dinitz's algorithm has a running time of $O(m\sqrt{n})$ on this class of networks and generates blocking flows in $O(m)$ time. Surprisingly, Tidal Flow is not that much slower than Dinitz's algorithm on these cases. Tidal Flow outperformed Dinitz's algorithm on dense level graphs and dense bipartite graphs with large edge capacities.

6.2. *Importance of the Heuristic Function.*

Two versions of Tidal Flow were implemented for the purpose of measuring the importance of the heuristic function h. In one implementation, the heuristic function h was removed. In every graph class, Tidal Flow performed significantly worse without the heuristic function. The effect of the heuristic function is to help identify bottlenecks in the network. When the heuristic function is tight, more flow can be sent down other paths in the network during low tide. The importance of the heuristic function was most pronounced in the grid network. This behavior is most likely due to the fact that level graphs formed from the grid network have a large amount of separating and rejoining paths. The heuristic function provides reasonable guidance so that less flow gets stuck in the middle of the network during low tide on such networks.

7. Concluding Remarks

This paper introduced the Tidal Flow algorithm and gave a preliminary survey measuring the performance against other flow algorithms. Tidal flow is both simpler to understand and implement than other fast flow algorithms. The relationship to preflows makes Tidal Flow a good intermediate algorithm for understanding more complicated algorithms like Preflow-Push. Though this paper described an initial exploration of the algorithm, there are still a number of unknowns with respect to the performance of Tidal Flow. Though Tidal Flow performs well on random networks against other flow algorithms, more extensive testing is required to determine its worst case behavior. Additionally, the theoretical worst case running time of $O(nm^2)$ may not be tight. Whether it is possible to create a level graph requiring more than $O(n/log(n))$ tide cycles to block the network is also unknown. Finally, it is unclear if a better h function exists for guiding the low tide decisions.

8. Acknowledgements

The author is grateful to Antony Stabile and Travis Meade for their contributions in developing and simplifying the initial version of Tidal Flow, the students of the UCF Programming Team for the continued feedback on how Tidal Flow is taught, and Lisa Soros and Brian Dean for feedback on a preliminary version of this paper.

References

Ahuja, R.K., Magnanti, T.L., Orlin, J.B. (1993). *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Boykov, Y., Kolmogorov, V. (2004). An experimental comparison of min- cut/max- flow algorithms for energy minimization in vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(9), 1124–1137.

Cherkassy, B.V., Goldberg, A.V. (1995). On implementing push-relabel method for the maximum flow problem. In: *Proceedings of the 4th International IPCO Conference on Integer Programming and Combinatorial Optimization*. London, UK, UK. Springer-Verlag, 157–171.

Dinitz, Y.A. (1970). Algorithm for solution of a problem of maximum flow in a network with power estimation. *Doklady Akademii Nauk SSSR*, 11, 1277–1280.

Dinitz, Y.A. (2006). Dinitz's algorithm: The original and even's version. In: *Theoretical Computer Science: Essays in Memory of Shimon Even*, volume 3895. Springer, Berlin, Heidelberg, 218–240.

Edmonds, J., Karp, R.M. (1972). Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2), 248–264.

Ford, L.R., Fulkerson, D.R. (1955). Maximal flow through a network. *Canadian Journal of Mathematics*, 8, 399–404.

Forišek, M., Stienová, M. (2013). *Explaining Algorithms Using Metaphors*. Springer, New York, NY, 1 edition.

Goldberg, A.V., Hed, S., Kaplan, H., Tarjan, R.E., Werneck, R.F. (2011). Maximum flows by incremental breadth-first search. In: Demetrescu, C. and Halldórsson, M. M. (Eds.), *Algorithms – ESA 2011*. Berlin, Heidelberg. Springer Berlin Heidelberg, 457–468.

Goldberg, A.V., Tarjan, R.E. (1988). A new approach to the maximum-ow problem. *J. ACM*, 35(4), 921–940.
Goldberg, A.V., Tarjan, R.E. (2014). Efficient maximum flow algorithms. *Communications of the ACM*, 57(8), 82–89.
Harris, T. E., Ross, F. (1955). Fundamentals of a method for evaluating rail net capacities. Technical report, Rand Corporation, Santa Monica, CA.
Ahuja, R.K., Kodialam, M., Mishra, A.K., Orlin, J. (1997). Computational investigations of maximum flow algorithms. 97, 509–542.
Karzanov, A.V. (1974). Determining a maximal flow in a network by the method of preflows. *Doklady Akademii Nauk SSSR*, 15(2), 434–437.
Lakoff, G., Johnson, M. (2003). *Metaphors We Live By.* University of Chicago press, Chicago, IL.
Lerner, J., Wagner, D., Zweig, K.A. (Eds.) (2009). *Algorithmics of Large and Complex Networks: Design, Analysis, and Simulation*. Springer-Verlag, Berlin, Heidelberg.
Orlin, J.B. (2013). Max flows in o(nm) time, or better. In: *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, STOC'13. New York, NY, USA. ACM, 765–774.
Orlin, J.B., Ahuja, R.K. (1987). *New Distance-Directed Algorithms for Maximum Flow and Parametric Maximum Flow Problems, Technical Report*. MIT, Cambridge, MA.
Sedgewick, R., Wayne, K. (2011). *Algorithms*. Addison-Wesley Professional, 4th edition.

M. Fontaine is the host of algorithms YouTube talk show, Algorithms Live! He is the cow artist for USA Computing Olympiad, creating the t-shirt design since 2014 and has been a USACO coach since 2017. He was an instructor at the University of Central Florida from 2014 to 2017 and a coach of UCF's ACM ICPC team from 2013 to 2017. He received his M.S. from UCF and was a research assistant at UCF's Institute for Simulation and Training from 2008 to 2013 studying problems in distributed game-based training.