Matching - Flow for competive programing

Duy Huynh - 2016

Contents

1 Hopcroft–Karp algorithm **1**
1.1 Augmenting paths . . . 1
1.2 Algorithm . . . 2
1.3 Analysis . . . 2
1.4 Comparison with other bipartite matching algorithms . . . 3
1.5 Non-bipartite graphs . . . 3
1.6 Pseudocode . . . 3
1.6.1 Explanation . . . 3
1.7 See also . . . 4
1.8 Notes . . . 4
1.9 References . . . 5

2 Blossom algorithm **6**
2.1 Augmenting paths . . . 6
2.2 Blossoms and contractions . . . 7
2.3 Finding an augmenting path . . . 10
2.3.1 Examples . . . 11
2.3.2 Analysis . . . 12
2.3.3 Bipartite matching . . . 13
2.3.4 Weighted matching . . . 13
2.4 References . . . 13

3 Hungarian algorithm **14**
3.1 Simple explanation of the assignment problem . . . 14
3.2 Setting . . . 14
3.3 The algorithm in terms of bipartite graphs . . . 15
3.4 Matrix interpretation . . . 15
3.5 Bibliography . . . 16
3.6 References . . . 16
3.7 External links . . . 17
3.7.1 Implementations . . . 17

4 Dinic’s algorithm **18**

4.1 History 18
4.2 Definition 18
4.3 Algorithm 19
4.4 Analysis 19
4.4.1 Special cases 19
4.5 Example 19
4.6 See also 20
4.7 Notes 20
4.8 References 20

5 Stable marriage problem **21**
5.1 Applications 21
5.2 Solution 22
5.3 Algorithm 23
5.4 Optimality of the solution 23
5.5 Stable Marriage with indifference 23
5.6 Similar problems 24
5.7 Implementation in software package 25
5.8 See also 25
5.9 References 25
5.9.1 Textbooks and other important references not cited in the text 26
5.10 External links 26

6 Independent set (graph theory) **27**
6.1 Properties 28
6.1.1 Relationship to other graph parameters 28
6.1.2 Maximal independent set 28
6.2 Finding independent sets 28
6.2.1 Maximum independent sets and maximum cliques 29
6.2.2 Finding maximum independent sets 29
6.2.3 Finding maximal independent sets 30
6.3 Software for searching maximum independent set 30
6.4 See also 30
6.5 Notes 31
6.6 References 31
6.7 External links 33

7 Maximal independent set **34**
7.1 Definition 35
7.2 Related vertex sets 36
7.3 Graph family characterizations 36
7.4 Bounding the number of sets 36

7.5 Finding a single maximal independent set . . . 37
7.5.1 Sequential algorithm . . . 37
7.5.2 Random-selection parallel algorithm . . . 37
7.5.3 Random-priority parallel algorithm . . . 38
7.5.4 Random-permutation parallel algorithm . . . 39
7.6 Listing all maximal independent sets . . . 40
7.7 Parallelization of finding maximum independent sets . . . 40
7.7.1 History . . . 40
7.7.2 Complexity class . . . 40
7.7.3 Communication and data exchange . . . 41
7.8 Notes . . . 41
7.9 References . . . 42

8 Minimum-cost flow problem **44**
8.1 Definition . . . 44
8.2 Relation to other problems . . . 44
8.3 Solutions . . . 45
8.4 Application . . . 45
8.4.1 Minimum weight bipartite matching . . . 45
8.5 See also . . . 46
8.6 References . . . 46
8.7 External links . . . 46

9 Max-flow min-cut theorem **47**
9.1 Definitions and statement . . . 47
9.1.1 Maximum flow . . . 47
9.1.2 Minimum cut . . . 47
9.1.3 Statement . . . 48
9.2 Linear program formulation . . . 48
9.3 Example . . . 48
9.4 Application . . . 49
9.4.1 Generalized max-flow min-cut theorem . . . 49
9.4.2 Menger's theorem . . . 49
9.4.3 Project selection problem . . . 49
9.4.4 Image segmentation problem . . . 50
9.5 History . . . 50
9.6 Proof . . . 50
9.7 See also . . . 51
9.8 References . . . 51

10 Edmonds–Karp algorithm **54**
10.1 Algorithm . . . 54

10.2 Pseudocode 54
10.3 Example 55
10.4 Notes 55
10.5 References 55
10.6 Text and image sources, contributors, and licenses 56
10.6.1 Text 56
10.6.2 Images 57
10.6.3 Content license 58

Chapter 1

Hopcroft–Karp algorithm

In computer science, the **Hopcroft–Karp algorithm** is an algorithm that takes as input a bipartite graph and produces as output a maximum cardinality matching – a set of as many edges as possible with the property that no two edges share an endpoint. It runs in $O(|E|\sqrt{|V|})$ time in the worst case, where E is set of edges in the graph, and V is set of vertices of the graph. In the case of dense graphs the time bound becomes $O(|V|^{2.5})$, and for random graphs it runs in near-linear time.

The algorithm was found by John Hopcroft and Richard Karp (1973). As in previous methods for matching such as the Hungarian algorithm and the work of Edmonds (1965), the Hopcroft–Karp algorithm repeatedly increases the size of a partial matching by finding augmenting paths. However, instead of finding just a single augmenting path per iteration, the algorithm finds a maximal set of shortest augmenting paths. As a result, only $O(\sqrt{|V|})$ iterations are needed. The same principle has also been used to develop more complicated algorithms for non-bipartite matching with the same asymptotic running time as the Hopcroft–Karp algorithm.

1.1 Augmenting paths

A vertex that is not the endpoint of an edge in some partial matching M is called a *free vertex*. The basic concept that the algorithm relies on is that of an *augmenting path*, a path that starts at a free vertex, ends at a free vertex, and alternates between unmatched and matched edges within the path. Note that except for the endpoints, all other vertices (if any) in augmenting path must be non-free vertices. An augmenting path could consist of only two vertices (both free) and single unmatched edge between them.

If M is a matching, and P is an augmenting path relative to M , then the symmetric difference of the two sets of edges, $M \oplus P$, would form a matching with size $|M|+1$. Thus, by finding augmenting paths, an algorithm may increase the size of the matching.

Conversely, suppose that a matching M is not optimal, and let P be the symmetric difference $M \oplus M^*$ where M^* is an optimal matching. Because M and M^* are both matchings, every vertex has degree at most 2 in P . So P must form a collection of disjoint augmenting paths and cycles or paths in which matched and unmatched edges are of equal number; the difference in size between M and M^* is the number of augmenting paths in P . Thus, if no augmenting path can be found, an algorithm may safely terminate, since in this case M must be optimal.

An augmenting path in a matching problem is closely related to the augmenting paths arising in maximum flow problems, paths along which one may increase the amount of flow between the terminals of the flow. It is possible to transform the bipartite matching problem into a maximum flow instance, such that the alternating paths of the matching problem become augmenting paths of the flow problem.[1] In fact, a generalization of the technique used in Hopcroft–Karp algorithm to arbitrary flow networks is known as Dinic's algorithm.

```
**Input**: Bipartite graph $G(U \cup V, E)$
**Output**: Matching $M \subseteq E$
$M \leftarrow \emptyset$
**repeat**
```

```
        $\mathcal{P} \leftarrow \{P_1, P_2, \ldots, P_k\}$ maximal set of vertex-disjoint shortest augmenting paths
        $M \leftarrow M \oplus (P_1 \cup P_2 \cup \cdots \cup P_k)$
    until $\mathcal{P} = \emptyset$
```

1.2 Algorithm

Let U and V be the two sets in the bipartition of G , and let the matching from U to V at any time be represented as the set M .

The algorithm is run in phases. Each phase consists of the following steps.

- A breadth-first search partitions the vertices of the graph into layers. The free vertices in U are used as the starting vertices of this search and form the first layer of the partitioning. At the first level of the search, there are only unmatched edges, since the free vertices in U are by definition not adjacent to any matched edges. At subsequent levels of the search, the traversed edges are required to alternate between matched and unmatched. That is, when searching for successors from a vertex in U , only unmatched edges may be traversed, while from a vertex in V only matched edges may be traversed. The search terminates at the first layer k where one or more free vertices in V are reached.
- All free vertices in V at layer k are collected into a set F . That is, a vertex v is put into F if and only if it ends a shortest augmenting path.
- The algorithm finds a maximal set of *vertex disjoint* augmenting paths of length k . This set may be computed by depth first search from F to the free vertices in U , using the breadth first layering to guide the search: the depth first search is only allowed to follow edges that lead to an unused vertex in the previous layer, and paths in the depth first search tree must alternate between matched and unmatched edges. Once an augmenting path is found that involves one of the vertices in F , the depth first search is continued from the next starting vertex.
- Every one of the paths found in this way is used to enlarge M .

The algorithm terminates when no more augmenting paths are found in the breadth first search part of one of the phases.

1.3 Analysis

Each phase consists of a single breadth first search and a single depth first search. Thus, a single phase may be implemented in linear time. Therefore, the first $\sqrt{|V|}$ phases, in a graph with $|V|$ vertices and $|E|$ edges, take time $O(|E|\sqrt{|V|})$.

It can be shown that each phase increases the length of the shortest augmenting path by at least one: the phase finds a maximal set of augmenting paths of the given length, so any remaining augmenting path must be longer. Therefore, once the initial $\sqrt{|V|}$ phases of the algorithm are complete, the shortest remaining augmenting path has at least $\sqrt{|V|}$ edges in it. However, the symmetric difference of the eventual optimal matching and of the partial matching M found by the initial phases forms a collection of vertex-disjoint augmenting paths and alternating cycles. If each of the paths in this collection has length at least $\sqrt{|V|}$, there can be at most $\sqrt{|V|}$ paths in the collection, and the size of the optimal matching can differ from the size of M by at most $\sqrt{|V|}$ edges. Since each phase of the algorithm increases the size of the matching by at least one, there can be at most $\sqrt{|V|}$ additional phases before the algorithm terminates.

Since the algorithm performs a total of at most $2\sqrt{|V|}$ phases, it takes a total time of $O(|E|\sqrt{|V|})$ in the worst case.

In many instances, however, the time taken by the algorithm may be even faster than this worst case analysis indicates. For instance, in the average case for sparse bipartite random graphs, Bast et al. (2006) (improving a previous result of Motwani 1994) showed that with high probability all non-optimal matchings have augmenting paths of logarithmic length. As a consequence, for these graphs, the Hopcroft–Karp algorithm takes $O(\log |V|)$ phases and $O(|E| \log |V|)$ total time.

1.4 Comparison with other bipartite matching algorithms

For sparse graphs, the Hopcroft–Karp algorithm continues to have the best known worst-case performance, but for dense graphs a more recent algorithm by Alt et al. (1991) achieves a slightly better time bound, $O\left(n^{1.5}\sqrt{\frac{m}{\log n}}\right)$. Their algorithm is based on using a push-relabel maximum flow algorithm and then, when the matching created by this algorithm becomes close to optimum, switching to the Hopcroft–Karp method.

Several authors have performed experimental comparisons of bipartite matching algorithms. Their results in general tend to show that the Hopcroft–Karp method is not as good in practice as it is in theory: it is outperformed both by simpler breadth-first and depth-first strategies for finding augmenting paths, and by push-relabel techniques.[2]

1.5 Non-bipartite graphs

The same idea of finding a maximal set of shortest augmenting paths works also for finding maximum cardinality matchings in non-bipartite graphs, and for the same reasons the algorithms based on this idea take $O(\sqrt{|V|})$ phases. However, for non-bipartite graphs, the task of finding the augmenting paths within each phase is more difficult. Building on the work of several slower predecessors, Micali & Vazirani (1980) showed how to implement a phase in linear time, resulting in a non-bipartite matching algorithm with the same time bound as the Hopcroft–Karp algorithm for bipartite graphs. The Micali–Vazirani technique is complex, and its authors did not provide full proofs of their results; subsequently, a "clear exposition" was published by Peterson & Loui (1988) and alternative methods were described by other authors.[3] In 2012, Vazirani offered a new simplified proof of the Micali-Vazirani algorithm.[4]

1.6 Pseudocode

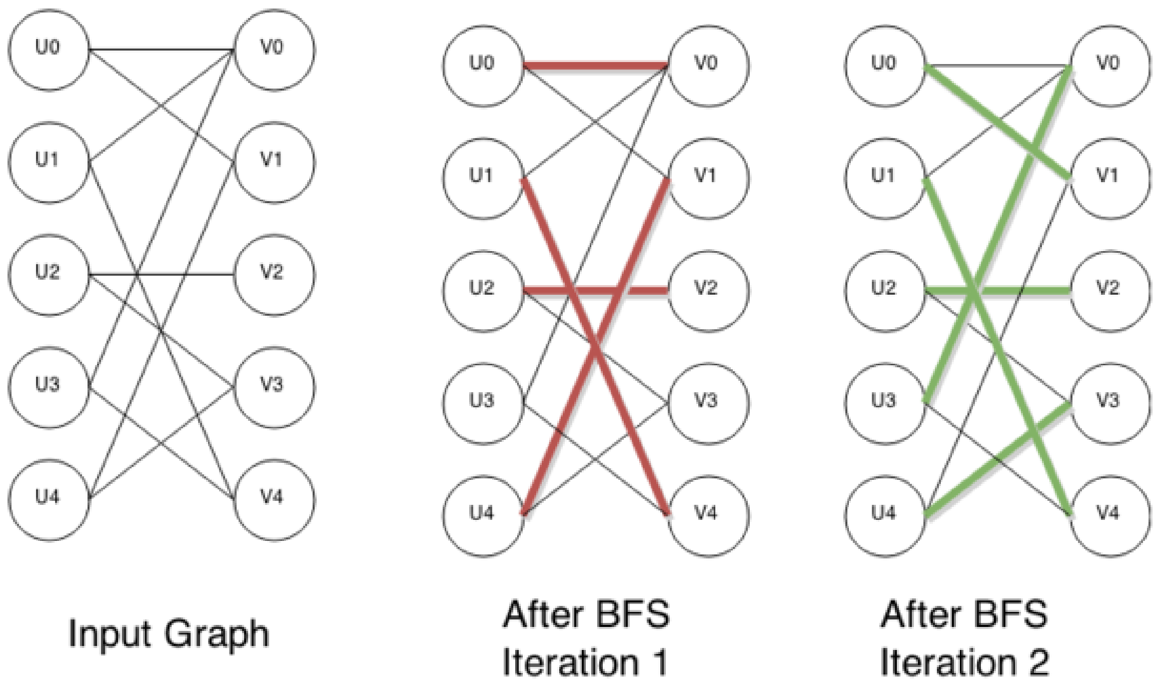
/* G = U ∪ V ∪ {NIL} where U and V are partition of graph and NIL is a special null vertex */ function BFS () for u in U if Pair_U[u] == NIL Dist[u] = 0 Enqueue(Q,u) else Dist[u] = ∞ Dist[NIL] = ∞ while Empty(Q) == false u = Dequeue(Q) if Dist[u] < Dist[NIL] for each v in Adj[u] if Dist[Pair_V[v]] == ∞ Dist[Pair_V[v]] = Dist[u] + 1 Enqueue(Q,Pair_V[v]) return Dist[NIL] != ∞ function DFS (u) if u != NIL for each v in Adj[u] if Dist[Pair_V[v]] == Dist[u] + 1 if DFS(Pair_V[v]) == true Pair_V[v] = u Pair_U[u] = v return true Dist[u] = ∞ return false return true function Hopcroft-Karp for each u in U Pair_U[u] = NIL for each v in V Pair_V[v] = NIL matching = 0 while BFS() == true for each u in U if Pair_U[u] == NIL if DFS(u) == true matching = matching + 1 return matching

1.6.1 Explanation

Let our graph have two partitions U, V. The key idea is to add two dummy vertices on each side of the graph: uDummy connecting to it to all unmatched vertices in U and vDummy connecting to all unmatched vertices in V. Now if we run BFS from uDummy to vDummy then we can get shortest path between an unmatched vertex in U to unmatched vertex in V. Due to bipartite nature of the graph, this path would zig zag from U to V. However we need to make sure that when going from V to U, we always select matched edge. If there is no matched edge then we end at vDummy. If we make sure of this criteria during BFS then the generated path would meet the criteria for being an augmented shortest path.

Once we have found the augmented shortest path, we want to make sure we ignore any other paths that are longer than this shortest paths. BFS algorithm marks nodes in path with distance with source being 0. Thus, after doing BFS, we can start at each unmatched vertex in U, follow the path by following nodes with distance that increments by 1. When we finally arrive at the destination vDummy, if its distance is 1 more than last node in V then we know that the path we followed is (one of the possibly many) shortest path. In that case we can go ahead and update the pairing for vertices on path in U and V. Note that each vertex in V on path, except for the last one, is non-free vertex. So updating pairing for these vertices in V to different vertices in U is equivalent to removing previously matched edge and adding new unmatched edge in matching. This is same as doing the symmetric difference (i.e. remove edges common to previous matching and add non-common edges in augmented path in new matching).

How do we make sure augmented paths are vertex disjoint? It is already guaranteed: After doing the symmetric difference for a path, none of its vertices could be considered again just because the Dist[Pair_V[v]] will not be equal to Dist[u] + 1 (It would be exactly Dist[u]).



Execution on an example graph showing input graph and matching after intermediate iteration 1 and final iteration 2.

So what is the mission of these two lines in pseudocode?:

Dist[u] = ∞ return false

When we were not able to find any shortest augmented path from a vertex, DFS returns false. In this case it would be good to mark these vertices to not to visit them again. This marking is simply done by setting Dist[u] to infinity.

Finally, we actually don't need uDummy because it’s there just to put all unmatched vertices of U in queue when BFS starts. That we can do as just as initialization. The vDummy can be appended in U for convenience in many implementations and initialize default pairing for all V to point to vDummy. That way, if final vertex in V doesn't have any matching vertex in U then we finally end at vDummy which is the end of our augmented path. In above pseudocode vDummy is denoted as Nil.

1.7 See also

- Bipartite matching
- Hungarian algorithm
- Assignment problem

1.8 Notes

[1] Ahuja, Magnanti & Orlin (1993), section 12.3, bipartite cardinality matching problem, pp. 469–470.

[2] Chang & McCormick (1990); Darby-Dowman (1980); Setubal (1993); Setubal (1996).

[3] Gabow & Tarjan (1991) and Blum (2001).

[4] Vazirani (2012)

1.9 References

- Ahuja, Ravindra K.; Magnanti, Thomas L.; Orlin, James B. (1993), *Network Flows: Theory, Algorithms and Applications*, Prentice-Hall.
- Alt, H.; Blum, N.; Mehlhorn, K.; Paul, M. (1991), "Computing a maximum cardinality matching in a bipartite graph in time $O\left(n^{1.5}\sqrt{\frac{m}{\log n}}\right)$ ", *Information Processing Letters*, **37** (4): 237–240, doi:10.1016/0020-0190(91)90195-N.
- Bast, Holger; Mehlhorn, Kurt; Schafer, Guido; Tamaki, Hisao (2006), "Matching algorithms are fast in sparse random graphs", *Theory of Computing Systems*, **39** (1): 3–14, doi:10.1007/s00224-005-1254-y.
- Blum, Norbert (2001), *A Simplified Realization of the Hopcroft-Karp Approach to Maximum Matching in General Graphs*, Tech. Rep. 85232-CS, Computer Science Department, Univ. of Bonn.
- Chang, S. Frank; McCormick, S. Thomas (1990), *A faster implementation of a bipartite cardinality matching algorithm*, Tech. Rep. 90-MSC-005, Faculty of Commerce and Business Administration, Univ. of British Columbia. As cited by Setubal (1996).
- Darby-Dowman, Kenneth (1980), *The exploitation of sparsity in large scale linear programming problems – Data structures and restructuring algorithms*, Ph.D. thesis, Brunel University. As cited by Setubal (1996).
- Edmonds, Jack (1965), "Paths, Trees and Flowers", *Canadian J. Math*, **17**: 449–467, doi:10.4153/CJM-1965-045-4, MR 0177907.
- Gabow, Harold N.; Tarjan, Robert E. (1991), "Faster scaling algorithms for general graph matching problems", *Journal of the ACM*, **38** (4): 815–853, doi:10.1145/115234.115366.
- Hopcroft, John E.; Karp, Richard M. (1973), "An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs", *SIAM Journal on Computing*, **2** (4): 225–231, doi:10.1137/0202019.
- Micali, S.; Vazirani, V. V. (1980), "An $O(\sqrt{|V|}\cdot|E|)$ algorithm for finding maximum matching in general graphs", *Proc. 21st IEEE Symp. Foundations of Computer Science*, pp. 17–27, doi:10.1109/SFCS.1980.12.
- Peterson, Paul A.; Loui, Michael C. (1988), "The general maximum matching algorithm of Micali and Vazirani", *Algorithmica*, **3** (1-4): 511–533, doi:10.1007/BF01762129.
- Motwani, Rajeev (1994), "Average-case analysis of algorithms for matchings and related problems", *Journal of the ACM*, **41** (6): 1329–1356, doi:10.1145/195613.195663.
- Setubal, João C. (1993), "New experimental results for bipartite matching", *Proc. Netflow93*, Dept. of Informatics, Univ. of Pisa, pp. 211–216. As cited by Setubal (1996).
- Setubal, João C. (1996), *Sequential and parallel experimental results with bipartite matching algorithms*, Tech. Rep. IC-96-09, Inst. of Computing, Univ. of Campinas.
- Vazirani, Vijay (2012), *An Improved Definition of Blossoms and a Simpler Proof of the MV Matching Algorithm*, CoRR abs/1210.4594, arXiv:1210.4594.

Chapter 2

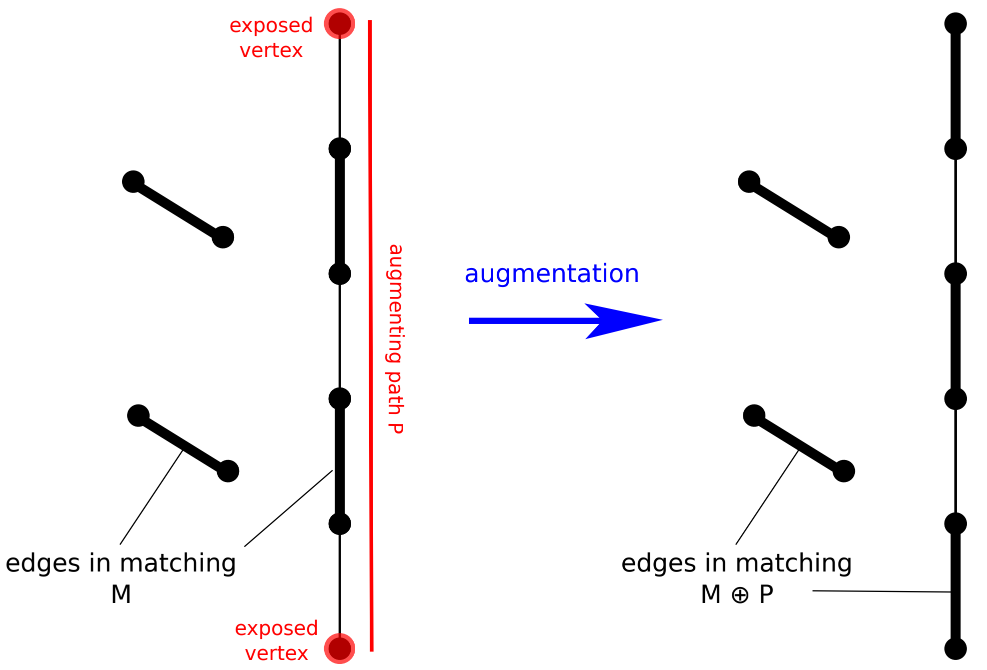
Blossom algorithm

The **blossom algorithm** is an algorithm in graph theory for constructing maximum matchings on graphs. The algorithm was developed by Jack Edmonds in 1961,[1] and published in 1965.[2] Given a general graph $G = (V, E)$, the algorithm finds a matching M such that each vertex in V is incident with at most one edge in M and $|M|$ is maximized. The matching is constructed by iteratively improving an initial empty matching along augmenting paths in the graph. Unlike bipartite matching, the key new idea is that an odd-length cycle in the graph (blossom) is contracted to a single vertex, with the search continuing iteratively in the contracted graph.

A major reason that the blossom algorithm is important is that it gave the first proof that a maximum-size matching could be found using a polynomial amount of computation time. Another reason is that it led to a linear programming polyhedral description of the matching polytope, yielding an algorithm for min-*weight* matching.[3] As elaborated by Alexander Schrijver, further significance of the result comes from the fact that this was the first polytope whose proof of integrality "does not simply follow just from total unimodularity, and its description was a breakthrough in polyhedral combinatorics."[4]

2.1 Augmenting paths

Given $G = (V, E)$ and a matching M of G, a vertex v is **exposed** if no edge of M is incident with v. A path in G is an **alternating path**, if its edges are alternately not in M and in M (or in M and not in M). An **augmenting path** P is an alternating path that starts and ends at two distinct exposed vertices. A **matching augmentation** along an augmenting path P is the operation of replacing M with a new matching $M_1 = M \oplus P = (M \setminus P) \cup (P \setminus M)$.



A matching M is maximum if and only if there is no M-augmenting path in G.[5][6] Hence, either a matching is maximum, or it can be augmented. Thus, starting from an initial matching, we can compute a maximum matching by augmenting the current matching with augmenting paths as long as we can find them, and return whenever no augmenting paths are left. We can formalize the algorithm as follows:

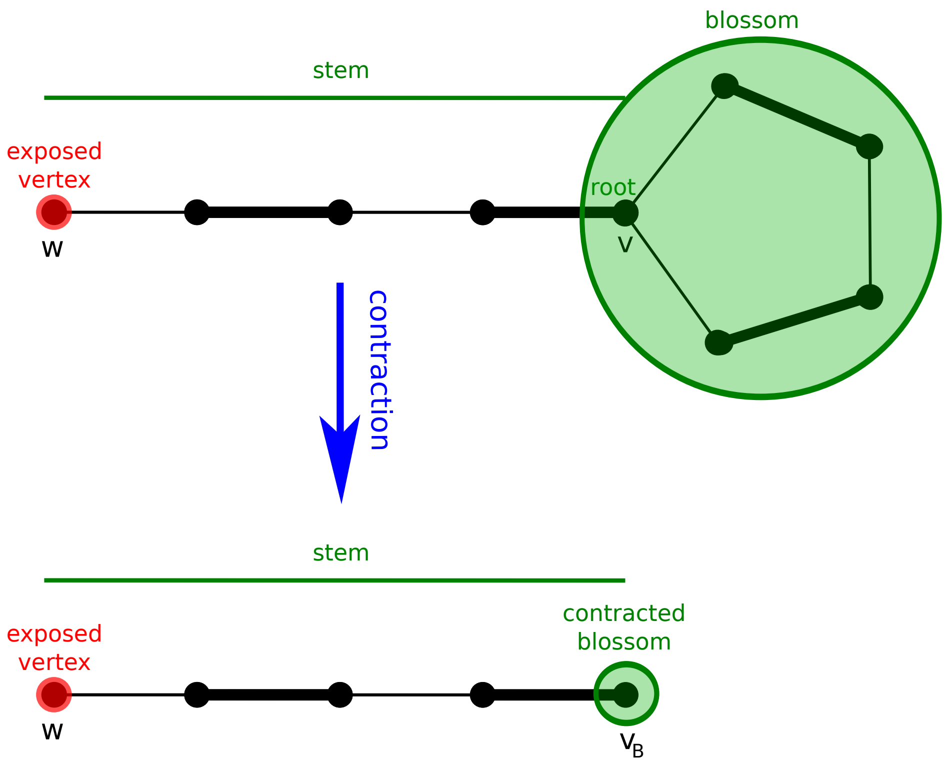
INPUT: Graph G, initial matching M on G OUTPUT: maximum matching M^* on G A1 **function** *find_maximum_matching*(G, M) : M^* A2 $P \leftarrow$ *find_augmenting_path*(G, M) A3 **if** P is non-empty **then** A4 **return** *find_maximum_matching*(G, augment M along P) A5 **else** A6 **return** M A7 **end if** A8 **end function**

We still have to describe how augmenting paths can be found efficiently. The subroutine to find them uses blossoms and contractions.

2.2 Blossoms and contractions

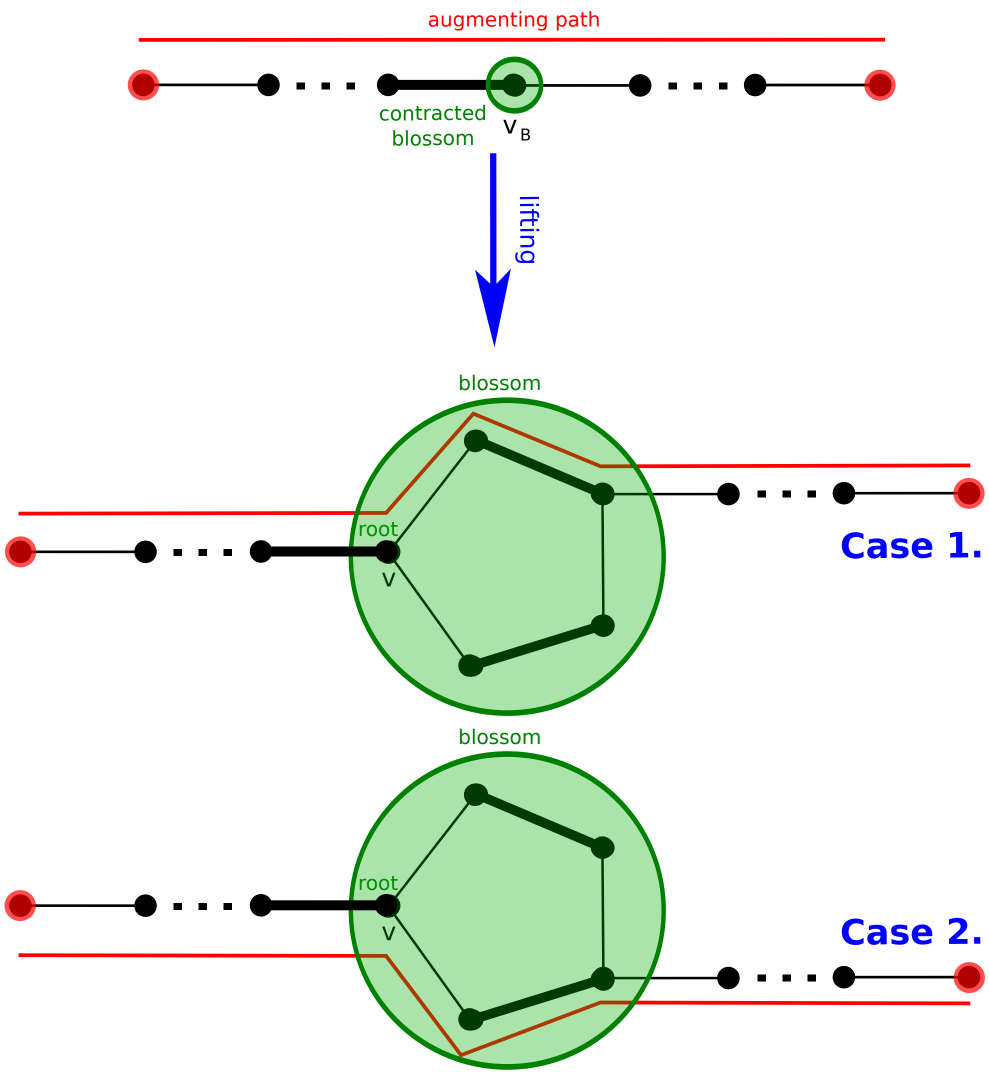
Given $G = (V, E)$ and a matching M of G, a *blossom* B is a cycle in G consisting of $2k + 1$ edges of which exactly k belong to M, and where one of the vertices v of the cycle (the *base*) is such that there exists an alternating path of even length (the *stem*) from v to an exposed vertex w.

Define the **contracted graph** G' as the graph obtained from G by contracting every edge of B, and define the **contracted matching** M' as the matching of G' corresponding to M.

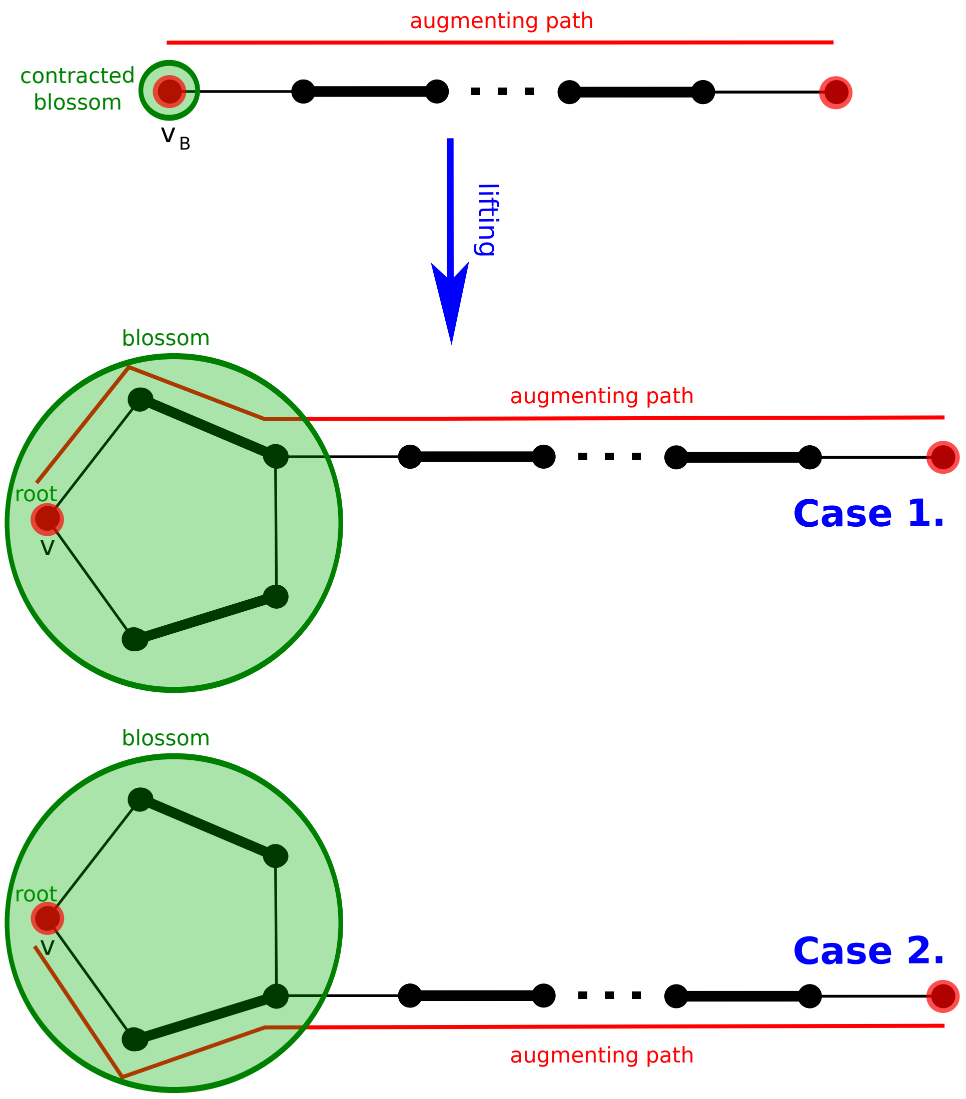


G' has an *M'*-augmenting path iff *G* has an *M*-augmenting path, and that any *M'*-augmenting path *P'* in *G'* can be **lifted** to an *M*-augmenting path in *G* by undoing the contraction by *B* so that the segment of *P'* (if any) traversing through *vB* is replaced by an appropriate segment traversing through *B*.[7] In more detail:

- if *P'* traverses through a segment *u* → *vB* → *w* in *G'*, then this segment is replaced with the segment *u* → (*u'* → ... → *w'*) → *w* in *G*, where blossom vertices *u'* and *w'* and the side of *B*, (*u'* → ... → *w'*), going from *u'* to *w'* are chosen to ensure that the new path is still alternating (*u'* is exposed with respect to $M \cap B$, $\{w', w\} \in E \setminus M$).



- if *P'* has an endpoint *vB*, then the path segment $u \rightarrow vB$ in *G'* is replaced with the segment $u \rightarrow$ (*u'* $\rightarrow$ ... $\rightarrow$ *v'*) in *G*, where blossom vertices *u'* and *v'* and the side of *B*, (*u'* $\rightarrow$ ... $\rightarrow$ *v'*), going from *u'* to *v'* are chosen to ensure that the path is alternating (*v'* is exposed, $\{u', u\} \in E \setminus M$).



Thus blossoms can be contracted and search performed in the contracted graphs. This reduction is at the heart of Edmonds' algorithm.

2.3 Finding an augmenting path

The search for an augmenting path uses an auxiliary data structure consisting of a forest F whose individual trees correspond to specific portions of the graph G. In fact, the forest F is the same that would be used to find maximum matchings in bipartite graphs (without need for shrinking blossoms). In each iteration the algorithm either (1) finds an augmenting path, (2) finds a blossom and recurses onto the corresponding contracted graph, or (3) concludes there are no augmenting paths. The auxiliary structure is built by an incremental procedure discussed next.[7]

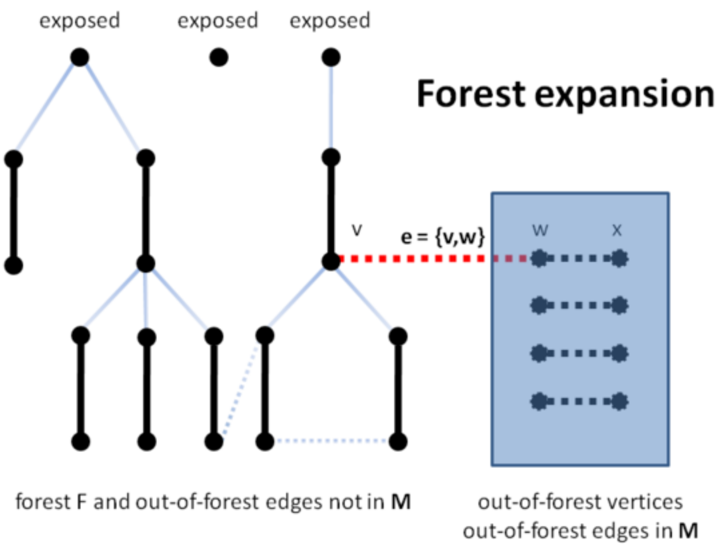
The construction procedure considers vertices v and edges e in G and incrementally updates F as appropriate. If v is in a tree T of the forest, we let *root(v)* denote the root of T. If both u and v are in the same tree T in F, we let *distance(u,v)* denote the length of the unique path from u to v in T.

INPUT: Graph G, matching M on G OUTPUT: augmenting path P in G or empty path if none found B01 **function** *find_augmenting_path*(G, M) : P B02 $F \leftarrow$ empty forest B03 unmark all vertices and edges in G, mark all edges of M B05 **for each** exposed vertex v **do** B06 create a singleton tree { v } and add the tree to F B07 **end for** B08 **while** there is an unmarked vertex v in F with *distance(v, root(v))* even **do** B09 **while** there exists an unmarked edge e = { v, w } **do** B10 **if** w is not in F **then** // w is matched, so add e and w's matched edge to F B11 $x \leftarrow$ vertex matched to w in M B12 add edges { v, w } and { w, x } to the tree of v B13 **else** B14 **if** *distance(w, root(w))* is odd **then** // Do nothing. B15 **else** B16 **if** *root(v)* $\neq$ *root(w)* **then** // Report an augmenting path in F $\cup$ { e }. B17 $P \leftarrow$ path

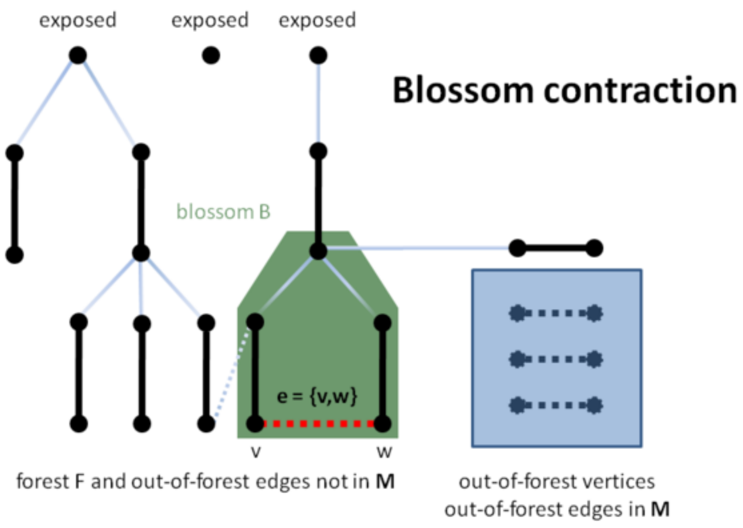
(*root*(*v*) → ... → *v*) → (*w* → ... → *root*(*w*)) B18 **return** *P* B19 **else** // Contract a blossom in *G* and look for the path in the contracted graph. B20 *B* ← blossom formed by *e* and edges on the path *v* → *w* in *T* B21 *G'*, *M'* ← contract *G* and *M* by *B* B22 *P'* ← *find_augmenting_path*(*G'*, *M'*) B23 *P* ← lift *P'* to *G* B24 **return** *P* B25 **end if** B26 **end if** B27 **end if** B28 mark edge *e* B29 **end while** B30 mark vertex *v* B31 **end while** B32 **return** empty path B33 **end function**

2.3.1 Examples

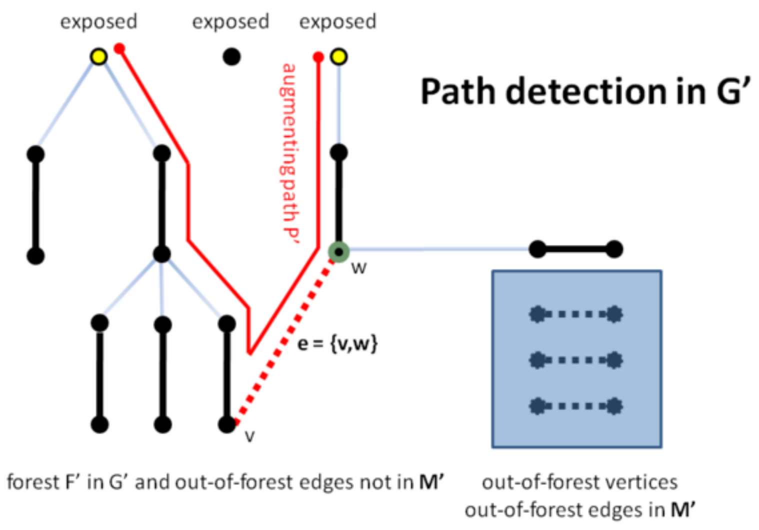
The following four figures illustrate the execution of the algorithm. Dashed lines indicate edges that are currently not present in the forest. First, the algorithm processes an out-of-forest edge that causes the expansion of the current forest (lines B10 – B12).

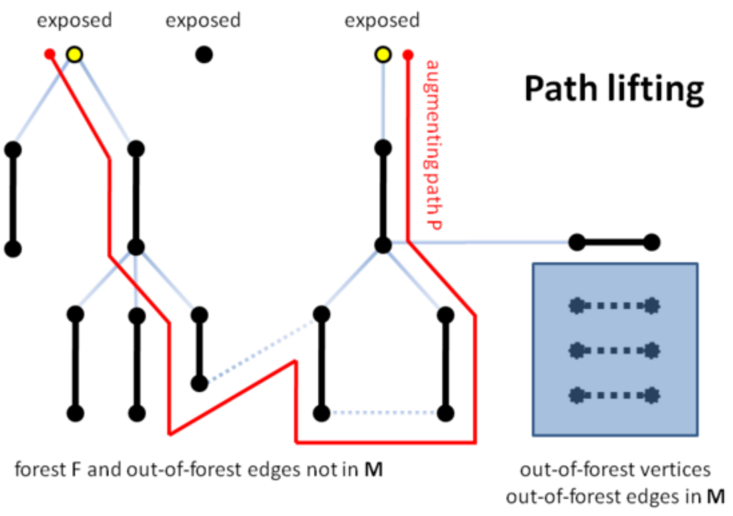


Next, it detects a blossom and contracts the graph (lines B20 – B21).



Finally, it locates an augmenting path P′ in the contracted graph (line B22) and lifts it to the original graph (line B23). Note that the ability of the algorithm to contract blossoms is crucial here; the algorithm cannot find *P* in the original graph directly because only out-of-forest edges between vertices at even distances from the roots are considered on line B17 of the algorithm.





2.3.2 Analysis

The forest F constructed by the *find_augmenting_path()* function is an alternating forest.[8]

- a tree T in G is an **alternating tree** with respect to M, if
  - T contains exactly one exposed vertex r called the tree root
  - every vertex at an odd distance from the root has exactly two incident edges in T, and
  - all paths from r to leaves in T have even lengths, their odd edges are not in M and their even edges are in M.
- a forest F in G is an **alternating forest** with respect to M, if
  - its connected components are alternating trees, and
  - every exposed vertex in G is a root of an alternating tree in F.

Each iteration of the loop starting at line B09 either adds to a tree T in F (line B10) or finds an augmenting path (line B17) or finds a blossom (line B20). It is easy to see that the running time is $O(|V|^4)$. Micali and Vazirani[9] show an algorithm that constructs maximum matching in $O(|E||V|^{1/2})$ time.

2.3.3 Bipartite matching

The algorithm reduces to the standard algorithm for matching in bipartite graphs[6] when G is bipartite. As there are no odd cycles in G in that case, blossoms will never be found and one can simply remove lines B20 – B24 of the algorithm.

2.3.4 Weighted matching

The matching problem can be generalized by assigning weights to edges in G and asking for a set M that produces a matching of maximum (minimum) total weight. The weighted matching problem can be solved by a combinatorial algorithm that uses the unweighted Edmonds's algorithm as a subroutine.[5] Kolmogorov provides an efficient C++ implementation of this.[10]

2.4 References

[1] Edmonds, Jack (1991), "A glimpse of heaven", in J.K. Lenstra; A.H.G. Rinnooy Kan; A. Schrijver, *History of Mathematical Programming --- A Collection of Personal Reminiscences*, CWI, Amsterdam and North-Holland, Amsterdam, pp. 32–54

[2] Edmonds, Jack (1965). "Paths, trees, and flowers". *Canad. J. Math.* **17**: 449–467. doi:10.4153/CJM-1965-045-4.

[3] Edmonds, Jack (1965). "Maximum matching and a polyhedron with 0,1-vertices". *Journal of Research of the National Bureau of Standards Section B*. **69**: 125–130.

[4] Schrijver, Alexander. *Combinatorial Optimization: Polyhedra and Efficiency*. Algorithms and Combinatorics. **24**. Springer.

[5] Lovász, László; Plummer, Michael (1986). *Matching Theory*. Akadémiai Kiadó. ISBN 963-05-4168-8.

[6] Karp, Richard, "Edmonds's Non-Bipartite Matching Algorithm", *Course Notes. U. C. Berkeley* (PDF)

[7] Tarjan, Robert, "Sketchy Notes on Edmonds' Incredible Shrinking Blossom Algorithm for General Matching", *Course Notes, Department of Computer Science, Princeton University* (PDF)

[8] Kenyon, Claire; Lovász, László, "Algorithmic Discrete Mathematics", *Technical Report CS-TR-251-90, Department of Computer Science, Princeton University*

[9] Micali, Silvio; Vazirani, Vijay (1980). *An $O(V^{1/2}E)$ algorithm for finding maximum matching in general graphs*. 21st Annual Symposium on Foundations of Computer Science,. IEEE Computer Society Press, New York. pp. 17–27.

[10] Kolmogorov, Vladimir (2009), "Blossom V: A new implementation of a minimum cost perfect matching algorithm", *Mathematical Programming Computation*, **1** (1): 43–67

Chapter 3

Hungarian algorithm

The **Hungarian method** is a combinatorial optimization algorithm that solves the assignment problem in polynomial time and which anticipated later primal-dual methods. It was developed and published in 1955 by Harold Kuhn, who gave the name “Hungarian method” because the algorithm was largely based on the earlier works of two Hungarian mathematicians: Dénes Kőnig and Jenő Egerváry.[1][2]

James Munkres reviewed the algorithm in 1957 and observed that it is (strongly) polynomial.[3] Since then the algorithm has been known also as the **Kuhn–Munkres algorithm** or **Munkres assignment algorithm**. The time complexity of the original algorithm was $O(n^4)$, however Edmonds and Karp, and independently Tomizawa noticed that it can be modified to achieve an $O(n^3)$ running time. Ford and Fulkerson extended the method to general transportation problems. In 2006, it was discovered that Carl Gustav Jacobi had solved the assignment problem in the 19th century, and the solution had been published posthumously in 1890 in Latin.[4]

3.1 Simple explanation of the assignment problem

In this simple example there are three workers: Jim, Steve, and Alan. One of them has to clean the bathroom, another sweep the floors, and the third wash the windows, but they each demand different pay for the various tasks. The problem is to find the lowest-cost way to assign the jobs. The problem can be represented in a matrix of the costs of the workers doing the jobs. For example:

The Hungarian method, when applied to the above table, would give the minimum cost: this is $6, achieved by having Jim clean the bathroom, Steve sweep the floors, and Alan wash the windows.

3.2 Setting

We are given a nonnegative *n*×*n* matrix, where the element in the *i*-th row and *j*-th column represents the cost of assigning the *j*-th job to the *i*-th worker. We have to find an assignment of the jobs to the workers that has minimum cost. If the goal is to find the assignment that yields the maximum cost, the problem can be altered to fit the setting by replacing each cost with the maximum cost subtracted by the cost.

The algorithm is easier to describe if we formulate the problem using a bipartite graph. We have a complete bipartite graph $G = (S, T; E)$ with n worker vertices (S) and n job vertices (T), and each edge has a nonnegative cost $c(i, j)$. We want to find a perfect matching with minimum cost.

Let us call a function $y : (S \cup T) \to \mathbb{R}$ a **potential** if $y(i) + y(j) \leq c(i, j)$ for each $i \in S, j \in T$. The value of potential y is $\sum_{v \in S \cup T} y(v)$. It can be seen that the cost of each perfect matching is at least the value of each potential. The Hungarian method finds a perfect matching and a potential with equal cost/value which proves the optimality of both. In fact it finds a perfect matching of **tight edges**: an edge ij is called tight for a potential y if $y(i) + y(j) = c(i, j)$. Let us denote the subgraph of tight edges by G_y . The cost of a perfect matching in G_y (if there is one) equals the value of y .

3.3 The algorithm in terms of bipartite graphs

During the algorithm we maintain a potential *y* and an orientation of G_y (denoted by $\overrightarrow{G_y}$) which has the property that the edges oriented from *T* to *S* form a matching *M*. Initially, *y* is 0 everywhere, and all edges are oriented from *S* to *T* (so *M* is empty). In each step, either we modify *y* so that its value increases, or modify the orientation to obtain a matching with more edges. We maintain the invariant that all the edges of *M* are tight. We are done if *M* is a perfect matching.

In a general step, let $R_S \subseteq S$ and $R_T \subseteq T$ be the vertices not covered by *M* (so R_S consists of the vertices in *S* with no incoming edge and R_T consists of the vertices in *T* with no outgoing edge). Let Z be the set of vertices reachable in $\overrightarrow{G_y}$ from R_S by a directed path only following edges that are tight. This can be computed by breadth-first search.

If $R_T \cap Z$ is nonempty, then reverse the orientation of a directed path in $\overrightarrow{G_y}$ from R_S to R_T . Thus the size of the corresponding matching increases by 1.

If $R_T \cap Z$ is empty, then let $\Delta := \min\{c(i,j) - y(i) - y(j) : i \in Z \cap S, j \in T \setminus Z\}$. Δ is positive because there are no tight edges between $Z \cap S$ and $T \setminus Z$. Increase *y* by Δ on the vertices of $Z \cap S$ and decrease *y* by Δ on the vertices of $Z \cap T$. The resulting *y* is still a potential. The graph G_y changes, but it still contains *M*. We orient the new edges from *S* to *T*. By the definition of Δ the set *Z* of vertices reachable from R_S increases (note that the number of tight edges does not necessarily increase).

We repeat these steps until *M* is a perfect matching, in which case it gives a minimum cost assignment. The running time of this version of the method is $O(n^4)$: *M* is augmented *n* times, and in a phase where *M* is unchanged, there are at most *n* potential changes (since *Z* increases every time). The time needed for a potential change is $O(n^2)$.

3.4 Matrix interpretation

Given n workers and tasks, and an *n*×*n* matrix containing the cost of assigning each worker to a task, find the cost minimizing assignment.

First the problem is written in the form of a matrix as given below

$$\begin{bmatrix} a1 & a2 & a3 & a4 \\ b1 & b2 & b3 & b4 \\ c1 & c2 & c3 & c4 \\ d1 & d2 & d3 & d4 \end{bmatrix}$$

where a, b, c and d are the workers who have to perform tasks 1, 2, 3 and 4. a1, a2, a3, a4 denote the penalties incurred when worker “a” does task 1, 2, 3, 4 respectively. The same holds true for the other symbols as well. The matrix is square, so each worker can perform only one task.

Step 1

Then we perform row operations on the matrix. To do this, the lowest of all *ai* (i belonging to 1-4) is taken and is subtracted from each element in that row. This will lead to at least one zero in that row (We get multiple zeros when there are two equal elements which also happen to be the lowest in that row). This procedure is repeated for all rows. We now have a matrix with at least one zero per row. Now we try to assign tasks to agents such that each agent is doing only one task and the penalty incurred in each case is zero. This is illustrated below.

The zeros that are indicated as 0' are the assigned tasks.

Step 2

Sometimes it may turn out that the matrix at this stage cannot be used for assigning, as is the case in for the matrix below.

In the above case, no assignment can be made. Note that task 1 is done efficiently by both agent a and c. Both can't be assigned the same task. Also note that no one does task 3 efficiently. To overcome this, we repeat the above procedure for all columns (i.e. the minimum element in each column is subtracted from all the elements in that column) and then check if an assignment is possible.

In most situations this will give the result, but if it is still not possible then we need to keep going.

Step 3

All zeros in the matrix must be covered by marking as few rows and/or columns as possible. The following procedure is one way to accomplish this:

First, assign as many tasks as possible.

- Row 1 has one zero, so it is assigned. The 0 in row 3 is crossed out because it is in the same column.
- Row 2 has one zero, so it is assigned.
- Row 3's only zero has been crossed out, so nothing is assigned.
- Row 4 has two uncrossed zeros. Either one can be assigned (both are optimum), and the other zero would be crossed out.

Alternatively, the 0 in row 3 may be assigned, causing the 0 in row 1 to be crossed instead.

Now to the drawing part.

- Mark all rows having no assignments (row 3).
- Mark all (unmarked) columns having zeros in newly marked row(s) (column 1).
- Mark all rows having assignments in newly marked columns (row 1).
- Repeat for all non-assigned rows.

Now draw lines through all marked columns and **unmarked** rows.

The aforementioned detailed description is just one way to draw the minimum number of lines to cover all the 0s. Other methods work as well.

Step 4

From the elements that are left, find the lowest value. Subtract this from every unmarked element and add it to every element covered by two lines.

Repeat steps 3–4 until an assignment is possible; this is when the minimum number of lines used to cover all the 0s is equal to the max(number of people, number of assignments), assuming dummy variables (usually the max cost) are used to fill in when the number of people is greater than the number of assignments.

Basically you find the second minimum cost among the remaining choices. The procedure is repeated until you are able to distinguish among the workers in terms of least cost.

3.5 Bibliography

- R.E. Burkard, M. Dell'Amico, S. Martello: *Assignment Problems* (Revised reprint). SIAM, Philadelphia (PA.) 2012. ISBN 978-1-61197-222-1
- M. Fischetti, "Lezioni di Ricerca Operativa", Edizioni Libreria Progetto Padova, Italia, 1995.
- R. Ahuja, T. Magnanti, J. Orlin, "Network Flows", Prentice Hall, 1993.
- S. Martello, "Jeno Egerváry: from the origins of the Hungarian algorithm to satellite communication". Central European Journal of Operations Research 18, 47–58, 2010

3.6 References

[1] Harold W. Kuhn, "The Hungarian Method for the assignment problem", *Naval Research Logistics Quarterly*, **2**: 83–97, 1955. Kuhn's original publication.

[2] Harold W. Kuhn, "Variants of the Hungarian method for assignment problems", *Naval Research Logistics Quarterly*, **3**: 253–258, 1956.

[3] J. Munkres, “Algorithms for the Assignment and Transportation Problems”, *Journal of the Society for Industrial and Applied Mathematics*, **5**(1):32–38, 1957 March.

[4] http://www.lix.polytechnique.fr/~{}ollivier/JACOBI/jacobiEngl.htm

3.7 External links

- Bruff, Derek, “The Assignment Problem and the Hungarian Method”,
- Mordecai J. Golin, Bipartite Matching and the Hungarian Method, Course Notes, Hong Kong University of Science and Technology.
- R. A. Pilgrim, *Munkres’ Assignment Algorithm. Modified for Rectangular Matrices*, Course notes, Murray State University.
- Mike Dawes, *The Optimal Assignment Problem*, Course notes, University of Western Ontario.
- On Kuhn’s Hungarian Method – A tribute from Hungary, András Frank, Egervary Research Group, Pazmany P. setany 1/C, H1117, Budapest, Hungary.
- Lecture: Fundamentals of Operations Research - Assignment Problem - Hungarian Algorithm, Prof. G. Srinivasan, Department of Management Studies, IIT Madras.
- Extension: Assignment sensitivity analysis (with O(n^4) time complexity), Liu, Shell.
- Solve any Assignment Problem online, provides a step by step explanation of the Hungarian Algorithm.

3.7.1 Implementations

(Note that not all of these satisfy the $O(n^3)$ time constraint.)

Chapter 4

Dinic's algorithm

Dinic's algorithm or **Dinitz's algorithm** is a strongly polynomial algorithm for computing the maximum flow in a flow network, conceived in 1970 by Israeli (formerly Soviet) computer scientist Yefim (Chaim) A. Dinitz.[1] The algorithm runs in $O(V^2E)$ time and is similar to the Edmonds–Karp algorithm, which runs in $O(VE^2)$ time, in that it uses shortest augmenting paths. The introduction of the concepts of the *level graph* and *blocking flow* enable Dinic's algorithm to achieve its performance.

4.1 History

Yefim Dinitz invented this algorithm in response to a pre-class exercise in Adel'son-Vel'sky's (co-inventor of AVL trees) Algorithm class. At the time he was not aware of the basic facts regarding Ford-Fulkerson algorithm.[2]

Dinitz mentions inventing his algorithm in January 1969, which was published in 1970 in journal *Doklady Akademii nauk SSSR*. In 1974, Shimon Even and (his then Ph.D. student) Alon Itai at the Technion in Haifa were very curious and intrigued by the Dinitz's algorithm as well as Alexander Karzanov's idea of blocking flow. However it was hard to decipher these two papers for them, each being limited to four pages to meet the restrictions of journal *Doklady Akademii nauk SSSR*. However Even did not give up and after three days of effort managed to understand both papers except for the layered network maintenance issue. Over the next couple of years, Even gave lectures on "Dinic's algorithm" mispronouncing the name of the author while popularizing it. Even and Itai also contributed to this algorithm by combining BFS and DFS, which is the current version of algorithm[3]

For about 10 years of time after Ford–Fulkerson algorithm was invented, it was unknown if it can be made to terminate in polynomial time in the generic case of irrational edge capacities. This caused lack of any known polynomial time algorithm that solved max flow problem in generic case. Dinitz algorithm and the Edmonds–Karp algorithm, which was published in 1972, independently showed that in the Ford–Fulkerson algorithm, if each augmenting path is the shortest one, the length of the augmenting paths is non-decreasing and it always terminated.

4.2 Definition

Let $G = ((V, E), c, s, t)$ be a network with $c(u, v)$ and $f(u, v)$ the capacity and the flow of the edge (u, v) respectively.

The **residual capacity** is a mapping $c_f \colon V \times V \to R^+$ defined as,

1. if $(u, v) \in E$,

   $c_f(u, v) = c(u, v) - f(u, v)$

   $c_f(v, u) = f(u, v)$

2. $c_f(u, v) = 0$ otherwise.

The **residual graph** is the graph $G_f = ((V, E_f), c_f|_{E_f}, s, t)$, where

$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$

An **augmenting path** is an $s - t$ path in the residual graph G_f .

Define dist(v) to be the length of the shortest path from s to v in G_f . Then the **level graph** of G_f is the graph $G_L = (V, E_L, c_f|_{E_L}, s, t)$, where

$E_L = \{(u, v) \in E_f : \text{dist}(v) = \text{dist}(u) + 1\}$

A **blocking flow** is an $s-t$ flow f such that the graph $G' = (V, E'_L, s, t)$ with $E'_L = \{(u, v) : f(u, v) < c_f|_{E_L}(u, v)\}$ contains no $s - t$ path.

4.3 Algorithm

Dinic's Algorithm

Input: A network $G = ((V, E), c, s, t)$.

Output: An $s - t$ flow f of maximum value.

1. Set $f(e) = 0$ for each $e \in E$.
2. Construct G_L from G_f of G . If dist$(t) = \infty$, stop and output f .
3. Find a blocking flow f' in G_L .
4. Augment flow f by f' and go back to step 2.

4.4 Analysis

It can be shown that the number of layers in each blocking flow increases by at least 1 each time and thus there are at most $n - 1$ blocking flows in the algorithm, where n is the number of vertices in the network. The level graph G_L can be constructed by Breadth-first search in $O(E)$ time and a blocking flow in each level graph can be found in $O(VE)$ time. Hence, the running time of Dinic's algorithm is $O(V^2 E)$.

Using a data structure called dynamic trees, the running time of finding a blocking flow in each phase can be reduced to $O(E \log V)$ and therefore the running time of Dinic's algorithm can be improved to $O(VE \log V)$.

4.4.1 Special cases

In networks with unit capacities, a much stronger time bound holds. Each blocking flow can be found in $O(E)$ time, and it can be shown that the number of phases does not exceed $O(\sqrt{E})$ and $O(V^{2/3})$. Thus the algorithm runs in $O(\min\{V^{2/3}, E^{1/2}\}E)$ time.[4]

In networks arising during the solution of bipartite matching problem, the number of phases is bounded by $O(\sqrt{V})$, therefore leading to the $O(\sqrt{V}E)$ time bound. The resulting algorithm is also known as Hopcroft–Karp algorithm. More generally, this bound holds for any *unit network* — a network in which each vertex, except for source and sink, either has a single entering edge of capacity one, or a single outgoing edge of capacity one, and all other capacities are arbitrary integers.[5]

4.5 Example

The following is a simulation of Dinic's algorithm. In the level graph G_L , the vertices with labels in red are the values dist(v) . The paths in blue form a blocking flow.

4.6 See also

- Ford–Fulkerson algorithm
- Maximum flow problem

4.7 Notes

[1] Yefim Dinitz (1970). "Algorithm for solution of a problem of maximum flow in a network with power estimation" (PDF). *Doklady Akademii nauk SSSR*. **11**: 1277–1280.

[2] Ilan Kadar; Sivan Albagli (2009-11-27). "Dinitz's algorithm for finding a maximum flow in a network".

[3] Yefim Dinitz. "Dinitz's Algorithm: The Original Version and Even's Version" (PDF).

[4] Even, Shimon; Tarjan, R. Endre (1975). "Network Flow and Testing Graph Connectivity". *SIAM Journal on Computing*. **4** (4): 507–518. doi:10.1137/0204043. ISSN 0097-5397.

[5] Tarjan 1983, p. 102.

4.8 References

- Yefim Dinitz (2006). "Dinitz' Algorithm: The Original Version and Even's Version" (PDF). In Oded Goldreich; Arnold L. Rosenberg; Alan L. Selman. *Theoretical Computer Science: Essays in Memory of Shimon Even*. Springer. pp. 218–240. ISBN 978-3-540-32880-3.
- Tarjan, R. E. (1983). *Data structures and network algorithms.*
- B. H. Korte; Jens Vygen (2008). "8.4 Blocking Flows and Fujishige's Algorithm". *Combinatorial Optimization: Theory and Algorithms (Algorithms and Combinatorics, 21)*. Springer Berlin Heidelberg. pp. 174–176. ISBN 978-3-540-71844-4.

Chapter 5

Stable marriage problem

In mathematics, economics, and computer science, the **stable marriage problem** (also **stable matching problem** or **SMP**) is the problem of finding a stable matching between two equally sized sets of elements given an ordering of preferences for each element. A matching is a mapping from the elements of one set to the elements of the other set. A matching is *not* stable if:

1. There is an element *A* of the first matched set which prefers some given element *B* of the second matched set over the element to which *A* is already matched, and

2. *B* also prefers *A* over the element to which *B* is already matched.

In other words, a matching is stable when there does not exist any match (*A*, *B*) by which *both A* and *B* would be individually better off than they are with the element to which they are currently matched.

The stable marriage problem has been stated as follows:

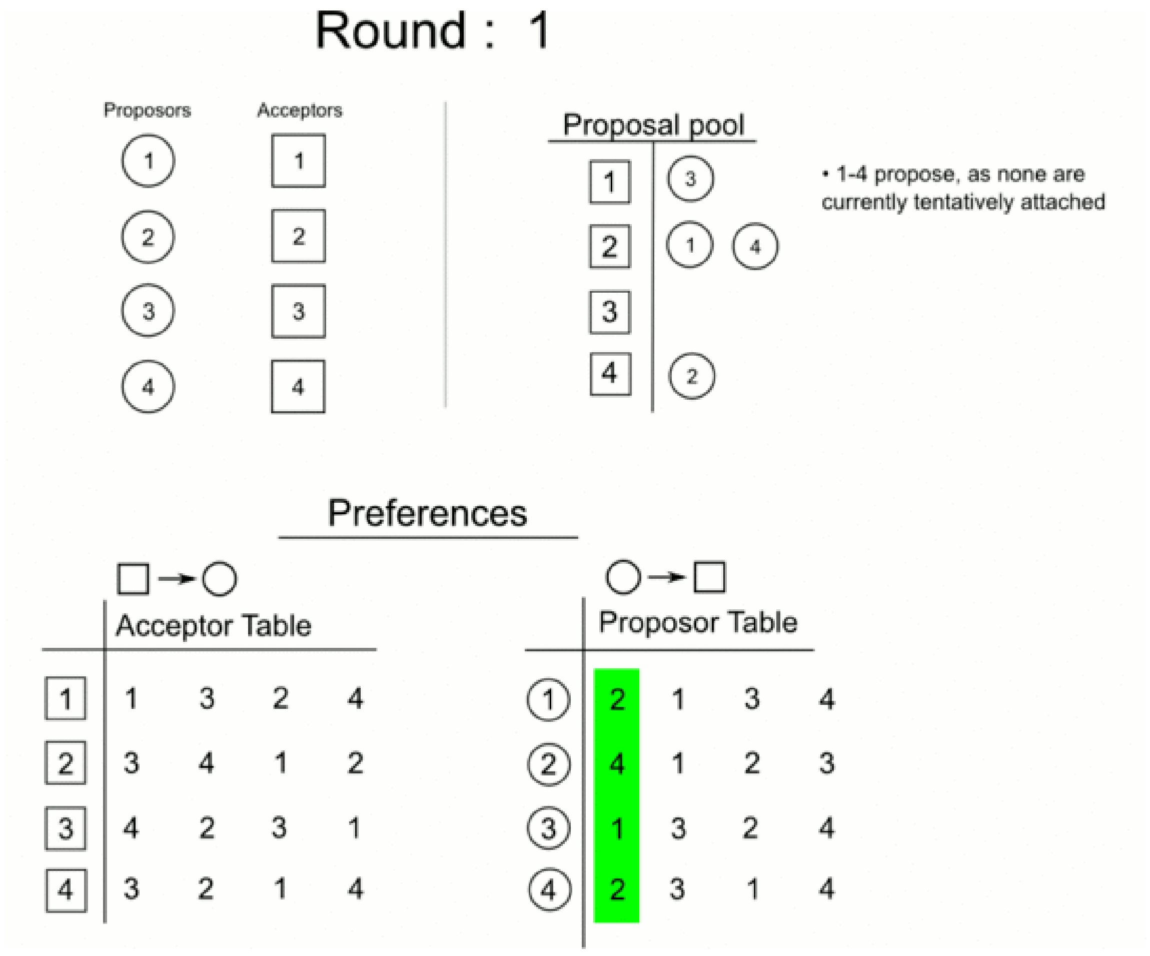
> Given *n* men and *n* women, where each person has ranked all members of the opposite sex in order of preference, marry the men and women together such that there are no two people of opposite sex who would both rather have each other than their current partners. When there are no such pairs of people, the set of marriages is deemed stable.

Note that the existence of two classes that need to be paired with each other (men and women in this example), distinguishes this problem from the stable roommates problem.

5.1 Applications

Algorithms for finding solutions to the stable marriage problem have applications in a variety of real-world situations, perhaps the best known of these being in the assignment of graduating medical students to their first hospital appointments.[1] In 2012, the Nobel Prize in Economics was awarded to Lloyd S. Shapley and Alvin E. Roth "for the theory of stable allocations and the practice of market design."[2]

An important and large-scale application of stable marriage is in assigning users to servers in a large distributed Internet service.[3] Billions of users access web pages, videos, and other services on the Internet, requiring each user to be matched to one of (potentially) hundreds of thousands of servers around the world that offer that service. A user prefers servers that are proximal enough to provide a faster response time for the requested service, resulting in a (partial) preferential ordering of the servers for each user. Each server prefers to serve users that it can with a lower cost, resulting in a (partial) preferential ordering of users for each server. Content delivery networks that distribute much of the world's content and services solve this large and complex stable marriage problem between users and servers every tens of seconds to enable billions of users to be matched up with their respective servers that can provide the requested web pages, videos, or other services.[3]



	Acceptor Table			
1	1	3	2	4
2	3	4	1	2
3	4	2	3	1
4	3	2	1	4

	Proposor Table			
1	2	1	3	4
2	4	1	2	3
3	1	3	2	4
4	2	3	1	4

Animation showing an example of the Gale–Shapley algorithm

5.2 Solution

In 1962, David Gale and Lloyd Shapley proved that, for any equal number of men and women, it is always possible to solve the SMP and make all marriages stable. They presented an algorithm to do so.[4][5]

The **Gale–Shapley algorithm** involves a number of “rounds” (or "iterations"). In the first round, first *a*) each unengaged man proposes to the woman he prefers most, and then *b*) each woman replies “maybe” to her suitor she most prefers and “no” to all other suitors. She is then provisionally “engaged” to the suitor she most prefers so far, and that suitor is likewise provisionally engaged to her. In each subsequent round, first *a*) each unengaged man proposes to the most-preferred woman to whom he has not yet proposed (regardless of whether the woman is already engaged), and then *b*) each woman replies “maybe” if she is currently not engaged or if she prefers this guy over her current provisional partner (in this case, she rejects her current provisional partner who becomes unengaged). The provisional nature of engagements preserves the right of an already-engaged woman to “trade up” (and, in the process, to “jilt” her until-then partner). This process is repeated until everyone is engaged.

The runtime complexity of this algorithm is $O(n^2)$ where n is number of men or women.[6]

This algorithm guarantees that:

Everyone gets married At the end, there cannot be a man and a woman both unengaged, as he must have proposed to her at some point (since a man will eventually propose to everyone, if necessary) and, being proposed to, she would necessarily be engaged (to someone) thereafter.

The marriages are stable Let Alice and Bob both be engaged, but not to each other. Upon completion of the algorithm, it is not possible for both Alice and Bob to prefer each other over their current partners. If Bob prefers Alice to his current partner, he must have proposed to Alice before he proposed to his current partner. If Alice accepted his proposal, yet is not married to him at the end, she must have dumped him for someone she likes more, and therefore doesn't like Bob more than her current partner. If Alice rejected his proposal, she was already with someone she liked more than Bob.

5.3 Algorithm

function stableMatching { Initialize all $m \in$ M and $w \in$ W to *free* **while** $\exists$ *free* man *m* who still has a woman w to propose to { w = first woman on m's list to whom m has not yet proposed **if** w is *free* (m, w) become *engaged* **else** some pair (m', w) already exists **if** w prefers m to m' m' becomes *free* (m, w) become *engaged* **else** (m', w) remain *engaged* } }

5.4 Optimality of the solution

While the solution is stable, it is not necessarily optimal from all individuals' points of view. The traditional form of the algorithm is optimal for the initiator of the proposals and the stable, suitor-optimal solution may or may not be optimal for the reviewer of the proposals. An example is as follows:

There are three suitors (A,B,C) and three reviewers (X,Y,Z) which have preferences of:

A: YXZ B: ZYX C: XZY X: BAC Y: CBA Z: ACB

There are 3 stable solutions to this matching arrangement:

suitors get their first choice and reviewers their third (AY, BZ, CX)

all participants get their second choice (AX, BY, CZ)

reviewers get their first choice and suitors their third (AZ, BX, CY)

All three are stable because instability requires both participants to be happier with an alternative match. Giving one group their first choices ensures that the matches are stable because they would be unhappy with any other proposed match. Giving everyone their second choice ensures that any other match would be disliked by one of the parties. The algorithm converges in a single round on the suitor-optimal solution because each reviewer receives exactly one proposal, and therefore selects that proposal as its best choice, ensuring that each suitor has an accepted offer, ending the match. This asymmetry of optimality is driven by the fact that the suitors have the entire set to choose from, but reviewers choose between a limited subset of the suitors at any one time.

5.5 Stable Marriage with indifference

In the classical version of the problem, each person must rank the members of the opposite sex in strict order of preference. However, in a real-world setting, a person may prefer two or more persons as equally favorable partner. Such tied preference is termed as indifference. Below is such an instance where m_2 finds tie between $w_3 \& w_1$ and w_2 finds tie between $m_1 \& m_2$.

$m_1[\, w_2\; w_1\; w_3\,]$ $w_1[\, m_3\; m_2\; m_1\,]$

$m_2[(w_3\; w_1)\, w_2]$ $w_2[(m_1\; m_2)\, m_3]$

$m_3[\, w_1\; w_2\; w_3\,]$ $w_3[\, m_2\; m_3\; m_1\,]$

If tied preference lists are allowed then the stable marriage problem will have three notions of stability which are discussed in the below sections.

A matching will be called **weakly stable** unless there is a couple each of whom strictly prefers the other to his/her partner in the matching. Robert W. Irving[7] has extended **Gale–Shapley algorithm** as below to provide such weakly stable matching in $O(n^2)$ time where n is size of stable marriage problem . Ties in men and women's preference list are broken arbitrarily. Preference lists are reduced as algorithm proceeds.

1 Assign each person to be free; 2 while (some man m is free) do 3 begin 4 w := first woman on m's list; 5 m proposes, and becomes engaged, to w; 6 if (some man m' is engaged to w) then 7 assign m' to be free; 8 for each (successor m''

of m on w’s list) do 9 delete the pair (m'', w) 10 end; 11 output the engaged pairs, which form a stable matching

A matching is **super-stable** if there is no couple each of whom either strictly prefers the other to his/her partner or is indifferent between them. Robert W. Irving[7] has modified the above algorithm to check whether such super stable matching exists and outputs matching in $O(n^2)$ time if it exists. Below is the pseudo-code.

1 assign each person to be free; 2 repeat 3 while (some man m is free) do 4 for each (woman w at the head of m’s list) do 5 begin 6 m proposes, and becomes engaged, to w; 7 for each (strict successor m' of m on w’s list) do 8 begin 9 if (m' is engaged) to w then 10 break the engagement; 11 delete the pair (m'. w) 12 end 13 end 14 for each (woman w who is multiply engaged) do 15 begin 16 break all engagements involving W; 17 for each (man m at the tail of w’s list) do 18 delete the pair (m. w) 19 end; 20 until (some man’s list is empty) or (everyone is engaged); 21 if everyone is engaged then 22 the engagement relation is a super-stable matching 23 else 24 no super-stable matching exists

A matching is **strongly stable** if there is no couple x, y such that x strictly prefers y to his/her partner and y either strictly prefers x to his/her partner or is indifferent between them. Robert W. Irving[7] has provided the algorithm which checks if such strongly stable matching exists and outputs the matching if exists. The algorithm computes perfect matching between set of men and women thus finding critical set of men who are engaged to multiple women. Since such engagements are never stable so all such pairs are deleted and proposal sequence will be repeated again until some man’s preference list becomes empty in which no strongly stable matching exists or strongly stable matching is obtained. Below is the pseudo-code finding strongly stable matching.It runs in $O(n^4)$ time which is explained in the Lemma 4.6 of

.[7]

1 Assign each person to be free; 2 repeat 3 while (some man m is free) do 4 for each (woman w at the head of m’s list) do 5 begin 6 m proposes, and becomes engaged, to w; 7 for each (strict successor m' of m on w’s list) do 8 begin 9 if (m' is engaged) to w then 10 break the engagement; 11 delete the pair (m'. w) 12 end 13 end 14 if (the engagement relation does not contain a perfect matching) then 15 begin 16 find the critical set Z of men; 17 for each (woman w who is engaged to a man in Z) do 18 begin 19 break all engagements involving w; 20 for each man m at the tail of w’s list do 21 delete the pair (m, w) 22 end; 23 end; 24 until (some man’s list is empty) or (everyone is engaged); 25 if everyone is engaged then 26 the engagement relation is a super-stable matching 27 else 28 no super-stable matching exists

5.6 Similar problems

The **assignment problem** seeks to find a matching in a weighted bipartite graph that has maximum weight. Maximum weighted matchings do not have to be stable, but in some applications a maximum weighted matching is better than a stable one.

The **stable roommates problem** is similar to the stable marriage problem, but differs in that all participants belong to a single pool (instead of being divided into equal numbers of “men” and “women”).

The **hospitals/residents problem** — also known as the **college admissions problem** — differs from the stable marriage problem in that the “women” can accept “proposals” from more than one “man” (e.g., a hospital can take multiple residents, or a college can take an incoming class of more than one student). Algorithms to solve the hospitals/residents problem can be *hospital-oriented* (female-optimal) or *resident-oriented* (male-optimal). This problem was solved, with an algorithm, in the same original paper by Gale and Shapley, in which the stable marriage problem was solved.[8]

The **hospitals/residents problem with couples** allows the set of residents to include couples who must be assigned together, either to the same hospital or to a specific pair of hospitals chosen by the couple (e.g., a married couple want to ensure that they will stay together and not be stuck in programs that are far away from each other). The addition of couples to the hospitals/residents problem renders the problem NP-complete.[9]

The matching with contracts problem is a generalization of matching problem, in which participants can be matched with different terms of contracts.[10] An important special case of contracts is matching with flexible wages.[11]

5.7 Implementation in software package

- R: The Gale-Shapley algorithm (also referred to as Deferred-Acceptance algorithm) for the stable marriage and the hospitals/residents problem is available as part of the matchingMarkets package.[12][13]

5.8 See also

- Assignment problem a similar problem where edge weights are commutative
- Stable roommates problem a similar problem, but with one set of size n and n-1 preferences
- Nash equilibrium
- Hungarian algorithm an algorithm to solve weighted bipartite matching problem
- Matching (graph theory) generalized matching problem in graphs
- Rainbow matching for edge colored graphs

5.9 References

[1] Stable Matching Algorithms

[2] "The Prize in Economic Sciences 2012". Nobelprize.org. Retrieved 2013-09-09.

[3] Bruce Maggs and Ramesh Sitaraman (2015). "Algorithmic nuggets in content delivery" (PDF). *ACM SIGCOMM Computer Communication Review*. **45** (3).

[4] Gale, D.; Shapley, L. S. (1962). "College Admissions and the Stability of Marriage". *American Mathematical Monthly*. **69**: 9–14. doi:10.2307/2312726. JSTOR 2312726.

[5] Harry Mairson: "The Stable Marriage Problem", *The Brandeis Review* 12, 1992 (online).

[6] Iwama, Kazuo; Miyazaki, Shuichi (2008). "A Survey of the Stable Marriage Problem and Its Variants". *International Conference on Informatics Education and Research for Knowledge-Circulating Society (icks 2008)*: 131–136. doi:10.1109/ICKS.2008.7.

[7] Irving, Robert W. (1994-02-15). "Stable marriage and indifference". *Discrete Applied Mathematics*. **48** (3): 261–272. doi:10.1016/0166-218X(92)00179-P.

[8] Gale, D.; Shapley, L. S. (1962). "College Admissions and the Stability of Marriage". *American Mathematical Monthly*. **69**: 9–14. doi:10.2307/2312726. JSTOR 2312726.

[9] Gusfield, D.; Irving, R. W. (1989). *The Stable Marriage Problem: Structure and Algorithms*. MIT Press. p. 54. ISBN 0-262-07118-5.

[10] Hatfield, John William; Milgrom, Paul (2005). "Matching with Contracts". *American Economic Review*. **95** (4): 913–935. doi:10.1257/0002828054825466. JSTOR 4132699.

[11] Crawford, Vincent; Knoer, Elsie Marie (1981). "Job Matching with Heterogeneous Firms and Workers". *Econometrica*. **49** (2): 437–450. doi:10.2307/1913320. JSTOR 1913320.

[12] Klein, T. (2015). "Analysis of Stable Matchings in R: Package matchingMarkets" (PDF). *Vignette to R Package matchingMarkets*.

[13] "matchingMarkets: Analysis of Stable Matchings". *R Project*.

5.9.1 Textbooks and other important references not cited in the text

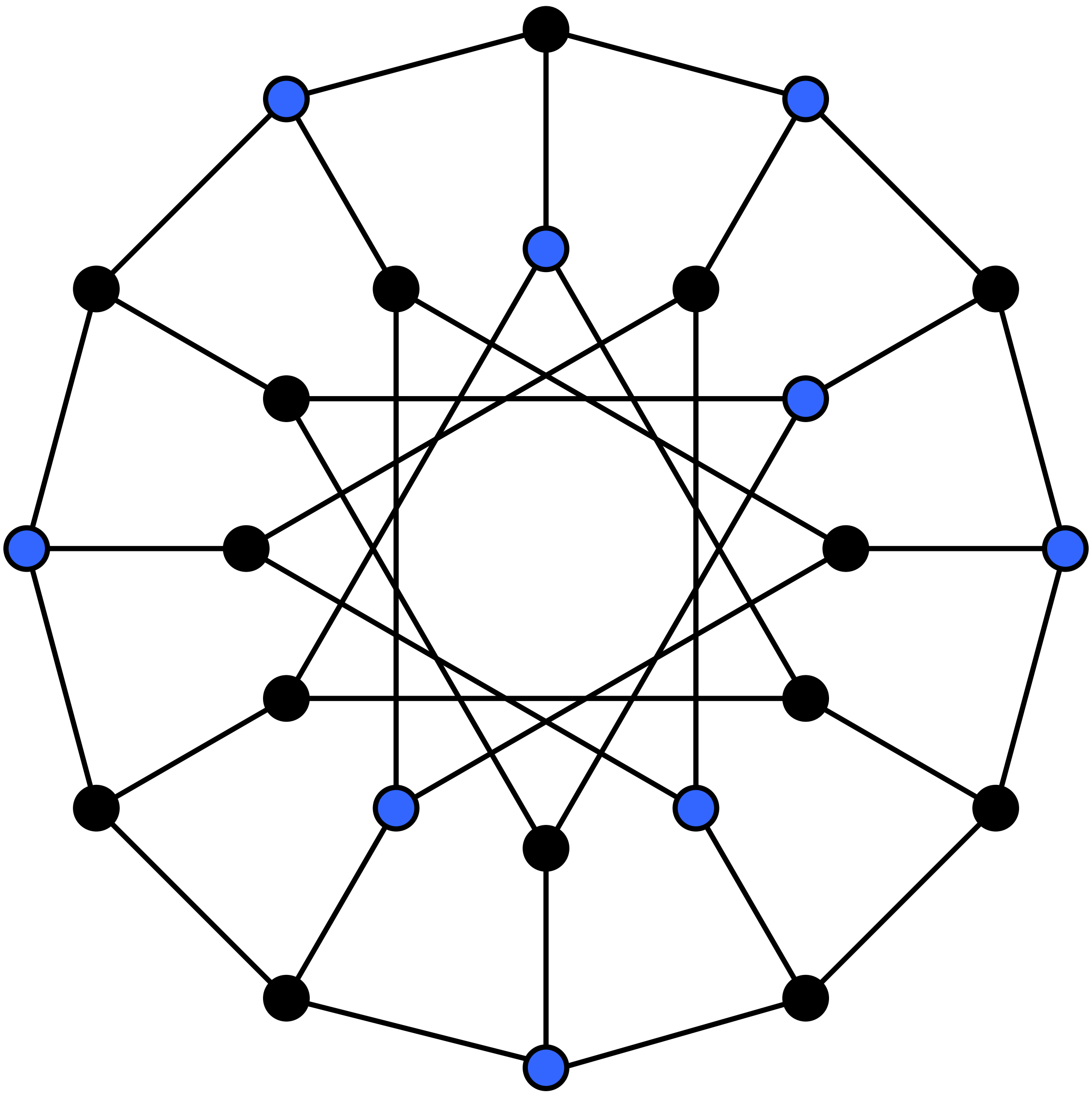
- Dubins, L.; Freedman, D. (1981). "Machiavelli and the Gale–Shapley algorithm". *American Mathematical Monthly*. **88** (7): 485–494. doi:10.2307/2321753.
- Kleinberg, J., and Tardos, E. (2005) *Algorithm Design*, Chapter 1, pp 1–12. See companion website for the Text .
- Knuth, D. E. (1976). *Mariages stables*. Montreal: Les Presses de I'Universite de Montreal.
- Knuth, D.E. (1996) *Stable Marriage and Its Relation to Other Combinatorial Problems: An Introduction to the Mathematical Analysis of Algorithms*, English translation, (CRM Proceedings and Lecture Notes), American Mathematical Society.
- Pittel, B. (1992). "On likely solutions of a stable marriage problem", The Annals of Applied Probability 2; 358-401.
- Roth, A. E. (1984). "The evolution of the labor market for medical interns and residents: A case study in game theory", Journal of Political Economy 92: 991–1016.
- Roth, A. E., and Sotomayor, M. A. O. (1990) *Two-sided matching: A study in game-theoretic modeling and analysis* Cambridge University Press.
- Shoham, Yoav; Leyton-Brown, Kevin (2009). *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. New York: Cambridge University Press. ISBN 978-0-521-89943-7. See Section 10.6.4; downloadable free online.
- Schummer, J.; Vohra, R. V. (2007). "Mechanism design without money". In Nisan, Noam; Roughgarden, Tim; Tardos, Eva; Vazirani, Vijay. *Algorithmic Game Theory* (PDF). pp. 255–262. ISBN 978-0521872829..

5.10 External links

- Interactive Flash Demonstration of SMP
- http://kuznets.fas.harvard.edu/~{ }aroth/alroth.html#NRMP
- http://www.dcs.gla.ac.uk/research/algorithms/stable/EGSapplet/EGS.html
- Gale–Shapley JavaScript Demonstration
- SMP Lecture Notes

Chapter 6

Independent set (graph theory)



The nine blue vertices form a maximum independent set for the Generalized Petersen graph GP(12,4).

In graph theory, an **independent set** or **stable set** is a set of vertices in a graph, no two of which are adjacent. That is, it is a set *S* of vertices such that for every two vertices in *S*, there is no edge connecting the two. Equivalently, each edge in the graph has at most one endpoint in *S*. The size of an independent set is the number of vertices it contains.

Independent sets have also been called internally stable sets.[1]

A maximal independent set is either an independent set such that adding any other vertex to the set forces the set to contain an edge or the set of all vertices of the empty graph.

A **maximum independent set** is an independent set of largest possible size for a given graph G. This size is called the **independence number** of G, and denoted $\alpha(G)$.[2] The problem of finding such a set is called the **maximum independent set problem** and is an NP-hard optimization problem. As such, it is unlikely that there exists an efficient algorithm for finding a maximum independent set of a graph.

Every maximum independent set also is maximal, but the converse implication does not necessarily hold.

6.1 Properties

6.1.1 Relationship to other graph parameters

A set is independent if and only if it is a clique in the graph's complement, so the two concepts are complementary. In fact, sufficiently large graphs with no large cliques have large independent sets, a theme that is explored in Ramsey theory.

A set is independent if and only if its complement is a vertex cover.[3] Therefore, the sum of the size of the largest independent set $\alpha(G)$, and the size of a minimum vertex cover $\beta(G)$, is equal to the number of vertices in the graph.

A vertex coloring of a graph G corresponds to a partition of its vertex set into independent subsets. Hence the minimal number of colors needed in a vertex coloring, the *chromatic number* $\chi(G)$, is at least the quotient of the number of vertices in G and the independent number $\alpha(G)$.

In a bipartite graph with no isolated vertices, the number of vertices in a maximum independent set equals the number of edges in a minimum edge covering; this is König's theorem.

6.1.2 Maximal independent set

Main article: Maximal independent set

An independent set that is not the subset of another independent set is called *maximal*. Such sets are dominating sets. Every graph contains at most $3^{n/3}$ maximal independent sets,[4] but many graphs have far fewer. The number of maximal independent sets in n-vertex cycle graphs is given by the Perrin numbers, and the number of maximal independent sets in n-vertex path graphs is given by the Padovan sequence.[5] Therefore, both numbers are proportional to powers of 1.324718, the plastic number.

6.2 Finding independent sets

Further information: Clique problem

In computer science, several computational problems related to independent sets have been studied.

- In the **maximum independent set** problem, the input is an undirected graph, and the output is a maximum independent set in the graph. If there are multiple maximum independent sets, only one need be output. This problem is sometimes referred to as "**vertex packing**".
- In the **maximum-weight independent set** problem, the input is an undirected graph with weights on its vertices and the output is an independent set with maximum total weight. The maximum independent set problem is the special case in which all weights are one.
- In the **maximal independent set listing** problem, the input is an undirected graph, and the output is a list of all its maximal independent sets. The maximum independent set problem may be solved using as a subroutine an algorithm for the maximal independent set listing problem, because the maximum independent set must be included among all the maximal independent sets.

- In the **independent set decision** problem, the input is an undirected graph and a number k, and the output is a Boolean value: true if the graph contains an independent set of size k, and false otherwise.

The first three of these problems are all important in practical applications; the independent set decision problem is not, but is necessary in order to apply the theory of NP-completeness to problems related to independent sets.

6.2.1 Maximum independent sets and maximum cliques

The independent set problem and the clique problem are complementary: a clique in G is an independent set in the complement graph of G and vice versa. Therefore, many computational results may be applied equally well to either problem. For example, the results related to the clique problem have the following corollaries:

- The independent set decision problem is NP-complete, and hence it is not believed that there is an efficient algorithm for solving it.
- The maximum independent set problem is NP-hard and it is also hard to approximate.

Despite the close relationship between maximum cliques and maximum independent sets in arbitrary graphs, the independent set and clique problems may be very different when restricted to special classes of graphs. For instance, for sparse graphs (graphs in which the number of edges is at most a constant times the number of vertices in any subgraph), the maximum clique has bounded size and may be found exactly in linear time;[6] however, for the same classes of graphs, or even for the more restricted class of bounded degree graphs, finding the maximum independent set is MAXSNP-complete, implying that, for some constant c (depending on the degree) it is NP-hard to find an approximate solution that comes within a factor of c of the optimum.[7]

6.2.2 Finding maximum independent sets

Further information: Clique problem § Finding maximum cliques in arbitrary graphs

Exact algorithms

The maximum independent set problem is NP-hard. However, it can be solved more efficiently than the $O(n^2\,2^n)$ time that would be given by a naive brute force algorithm that examines every vertex subset and checks whether it is an independent set.

An algorithm of Robson (1986) solves the problem in time $O(1.2108^n)$ using exponential space. When restricted to polynomial space, there is a time $O(1.2127^n)$ algorithm[8] which improves upon a simpler $O(1.2209^n)$ algorithm.[9]

For many classes of graphs, a maximum weight independent set may be found in polynomial time. Famous examples are claw-free graphs,[10] P_5-free graphs[11] and perfect graphs.[12] For chordal graphs, a maximum weight independent set can be found in linear time.[13]

Modular decomposition is a good tool for solving the maximum weight independent set problem; the linear time algorithm on cographs is the basic example for that. Another important tool are clique separators as described by Tarjan.[14]

In a bipartite graph, all nodes that are not in the minimum vertex cover can be included in maximum independent set; see König's theorem. Therefore, minimum vertex covers can be found using a bipartite matching algorithm.

Approximation algorithms

In general, the maximum independent set problem cannot be approximated to a constant factor in polynomial time (unless P = NP). In fact, Max Independent Set in general is Poly-APX-complete, meaning it is as hard as any problem that can be approximated to a polynomial factor.[15] However, there are efficient approximation algorithms for restricted classes of graphs.

In planar graphs, the maximum independent set may be approximated to within any approximation ratio $c < 1$ in polynomial time; similar polynomial-time approximation schemes exist in any family of graphs closed under taking minors.[16]

In bounded degree graphs, effective approximation algorithms are known with approximation ratios that are constant for a fixed value of the maximum degree; for instance, a greedy algorithm that forms a maximal independent set by, at each step, choosing the minimum degree vertex in the graph and removing its neighbors, achieves an approximation ratio of $(\Delta+2)/3$ on graphs with maximum degree Δ.[17] Approximation hardness bounds for such instances were proven in Berman & Karpinski (1999). Indeed, even Max Independent Set on 3-regular 3-edge-colorable graphs is APX-complete.[18]

Independent sets in interval intersection graphs

Main article: Interval scheduling

An interval graph is a graph in which the nodes are 1-dimensional intervals (e.g. time intervals) and there is an edge between two intervals iff they intersect. An independent set in an interval graph is just a set of non-overlapping intervals. The problem of finding maximum independent sets in interval graphs has been studied, for example, in the context of job scheduling: given a set of jobs that has to be executed on a computer, find a maximum set of jobs that can be executed without interfering with each other. This problem can be solved exactly in polynomial time using earliest deadline first scheduling.

Independent sets in geometric intersection graphs

Main article: Maximum disjoint set

A geometric intersection graph is a graph in which the nodes are geometric shapes and there is an edge between two shapes iff they intersect. An independent set in a geometric intersection graph is just a set of disjoint (non-overlapping) shapes. The problem of finding maximum independent sets in geometric intersection graphs has been studied, for example, in the context of Automatic label placement: given a set of locations in a map, find a maximum set of disjoint rectangular labels near these locations.

Finding a maximum independent set in intersection graphs is still NP-complete, but it is easier to approximate than the general maximum independent set problem. A recent survey can be found in the introduction of Chan & Har-Peled (2012).

6.2.3 Finding maximal independent sets

Main article: Maximal independent set

The problem of finding a maximal independent set can be solved in polynomial time by a trivial greedy algorithm.[19] All maximal independent sets can be found in time $O(3^{n/3}) = O(1.4423^n)$.

6.3 Software for searching maximum independent set

6.4 See also

- An independent set of edges is a set of edges of which no two have a vertex in common. It is usually called a matching.
- A vertex coloring is a partition of the vertex set into independent sets.

6.5 Notes

[1] Korshunov (1974)

[2] Godsil & Royle (2001), p. 3.

[3] PROOF: A set V of vertices is an independent set IFF every edge in the graph is adjacent to at most one member of V IFF every edge in the graph is adjacent to at least one member not in V IFF the complement of V is a vertex cover.

[4] Moon & Moser (1965).

[5] Füredi (1987).

[6] Chiba & Nishizeki (1985).

[7] Berman & Fujito (1995).

[8] Bourgeois et al. (2010)

[9] Fomin, Grandoni & Kratsch (2009)

[10] Minty (1980),Sbihi (1980),Nakamura & Tamura (2001),Faenza, Oriolo & Stauffer (2014),Nobili & Sassano (2015)

[11] Lokshtanov, Vatshelle & Villanger (2014)

[12] Grötschel, Lovász & Schrijver (1988)

[13] Frank (1976)

[14] Tarjan (1985)

[15] Bazgan, Cristina; Escoffier, Bruno; Paschos, Vangelis Th. (2005). "Completeness in standard and differential approximation classes: Poly-(D)APX- and (D)PTAS-completeness". *Theoretical Computer Science*. **339** (2–3): 272–292. doi:10.1016/j.tcs.2005.03.007.

[16] Baker (1994); Grohe (2003).

[17] Halldórsson & Radhakrishnan (1997).

[18] Chlebík, Miroslav; Chlebíková, Janka (2003). "Approximation Hardness for Small Occurrence Instances of NP-Hard Problems". *Proceedings of the 5th International Conference on Algorithms and Complexity*.

[19] Luby (1986).

6.6 References

- Baker, Brenda S. (1994), "Approximation algorithms for NP-complete problems on planar graphs", *Journal of the ACM*, **41** (1): 153–180, doi:10.1145/174644.174650.
- Berman, Piotr; Fujito, Toshihiro (1995), "On approximation properties of the Independent set problem for degree 3 graphs", *Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science, **955**, Springer-Verlag, pp. 449–460, doi:10.1007/3-540-60220-8_84, ISBN 978-3-540-60220-0.
- Berman, Piotr; Karpinski, Marek (1999), "On some tighter inapproximability results", *Automata, Languages and Programming, 26th International Colloquium, ICALP'99 Prague*, Lecture Notes in Computer Science, **1644**, Prague: Springer-Verlag, pp. 200–209, doi:10.1007/3-540-48523-6, ISBN 978-3-540-66224-2
- Bourgeois, Nicolas; Escoffier, Bruno; Paschos, Vangelis Th.; van Rooij, Johan M. M. (2010), "Algorithm theory—SWAT 2010", *Algorithm Theory – SWAT 2010*, Lecture Notes in Computer Science, Berlin: Springer, **6139**: 62–73, Bibcode:2010LNCS.6139...62B, doi:10.1007/978-3-642-13731-0_7, ISBN 978-3-642-13730-3, MR 2678485 |contribution= ignored (help).
- Chan, T. M. (2003), "Polynomial-time approximation schemes for packing and piercing fat objects", *Journal of Algorithms*, **46** (2): 178–189, doi:10.1016/s0196-6774(02)00294-8.
- Chan, T. M.; Har-Peled, S. (2012), "Approximation algorithms for maximum independent set of pseudo-disks", *Discrete & Computational Geometry*, **48** (2): 373, doi:10.1007/s00454-012-9417-5.

- Chiba, N.; Nishizeki, T. (1985), "Arboricity and subgraph listing algorithms", *SIAM Journal on Computing*, **14** (1): 210–223, doi:10.1137/0214017.
- Erlebach, T.; Jansen, K.; Seidel, E. (2005), "Polynomial-Time Approximation Schemes for Geometric Intersection Graphs", *SIAM Journal on Computing*, **34** (6): 1302, doi:10.1137/s0097539702402676.
- Faenza, Y.; Oriolo, G.; Stauffer, G. (2014), "Solving the Weighted Stable Set Problem in Claw-Free Graphs", *Journal of the ACM*, **61** (4): 1–41, doi:10.1145/2629600.
- Fomin, Fedor V.; Grandoni, Fabrizio; Kratsch, Dieter (2009), "A measure & conquer approach for the analysis of exact algorithms", *Journal of ACM*, **56** (5): 1–32, doi:10.1145/1552285.1552286, article no. 25.
- Frank, Andras (1976), "Some polynomial algorithms for certain graphs and hypergraphs", *Congressus Numerantium*, **XV**: 211–226.
- Füredi, Z. (1987), "The number of maximal independent sets in connected graphs", *Journal of Graph Theory*, **11** (4): 463–470, doi:10.1002/jgt.3190110403.
- Godsil, Chris; Royle, Gordon (2001), *Algebraic Graph Theory*, New York: Springer, ISBN 0-387-95220-9.
- Grohe, Martin (2003), "Local tree-width, excluded minors, and approximation algorithms", *Combinatorica*, **23** (4): 613–632, doi:10.1007/s00493-003-0037-9.
- Grötschel, M.; Lovász, L.; Schrijver, A. (1988), "9.4 Coloring Perfect Graphs", *Geometric Algorithms and Combinatorial Optimization*, Algorithms and Combinatorics, **2**, Springer-Verlag, pp. 296–298, ISBN 0-387-13624-X.
- Halldórsson, M. M.; Radhakrishnan, J. (1997), "Greed is good: Approximating independent sets in sparse and bounded-degree graphs", *Algorithmica*, **18** (1): 145–163, doi:10.1007/BF02523693.
- Korshunov, A.D. (1974), "Coefficient of Internal Stability", *Kibernetika* (in Ukrainian), **10** (1): 17–28, doi:10.1007/BF01069014.
- Lokshtanov, D.; Vatshelle, M.; Villanger, Y. (2014), "Independent sets in P_5-free graphs in polynomial time", *SODA (Symposium on Discrete Algorithms)*: 570–581.
- Luby, Michael (1986), "A simple parallel algorithm for the maximal independent set problem", *SIAM Journal on Computing*, **15** (4): 1036–1053, doi:10.1137/0215074, MR 861369.
- Minty, G.J. (1980), "On maximal independent sets of vertices in claw-free graphs", *Journal of Combinatorial Theory Series B*, **28**: 284–304, doi:10.1016/0095-8956(80)90074-x.
- Moon, J.W.; Moser, Leo (1965), "On cliques in graphs", *Israel Journal of Mathematics*, **3** (1): 23–28, doi:10.1007/BF02760024, MR 0182577.
- Nakamura, D.; Tamura, A. (2001), "A revision of Minty's algorithm for finding a maximum weight stable set in a claw-free graph", *Journal of Operations Research Society Japan* , **44**: 194–204.
- Nobili, P.; Sassano, A. (2015), *An O(n^2 log n) algorithm for the weighted stable set problem in claw-free graphs*, arXiv:1501.05775 [cs.DM]
- Robson, J. M. (1986), "Algorithms for maximum independent sets", *Journal of Algorithms*, **7** (3): 425–440, doi:10.1016/0196-6774(86)90032-5.
- Sbihi, Najiba (1980), "Algorithme de recherche d'un stable de cardinalité maximum dans un graphe sans étoile", *Discrete Mathematics* (in French), **29** (1): 53–76, doi:10.1016/0012-365X(90)90287-R, MR 553650.
- Tarjan, R.E. (1985), "Decomposition by clique separators", *Discrete Mathematics*, **55**: 221–232, doi:10.1016/0012-365x(85)90051-2.

6.7 External links

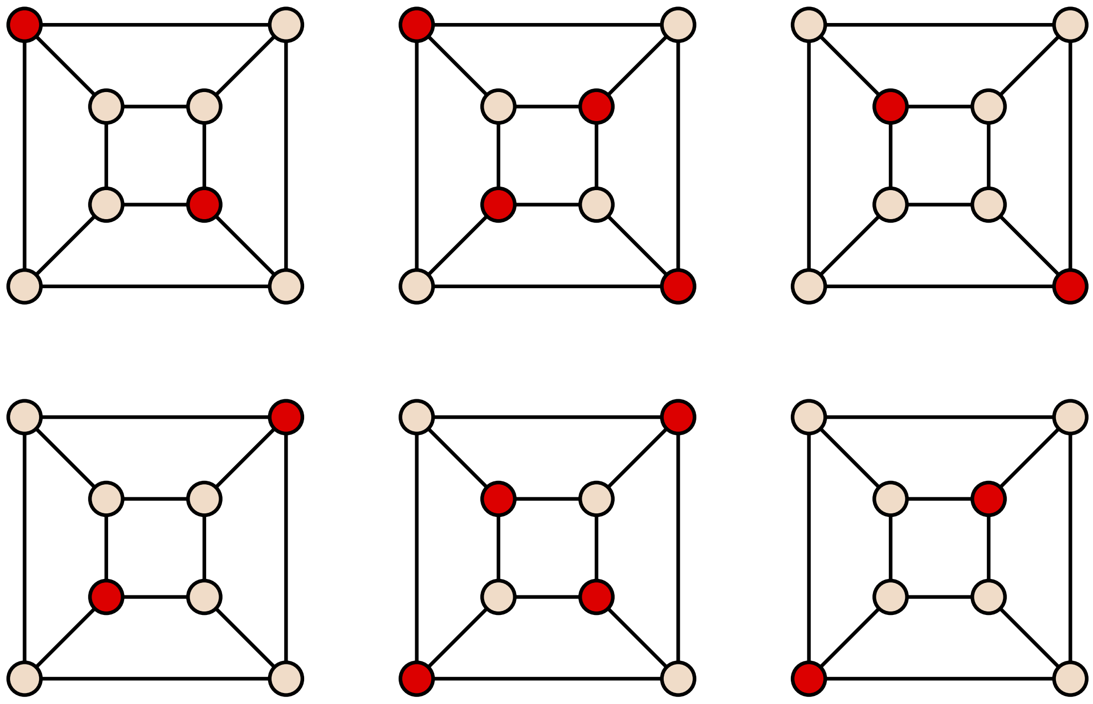
- Weisstein, Eric W. “Maximal Independent Vertex Set”. *MathWorld*.
- Challenging Benchmarks for Maximum Clique, Maximum Independent Set, Minimum Vertex Cover and Vertex Coloring
- Independent Set and Vertex Cover, Hanan Ayad.
- Maximum Independent Sets in Graphs – an interactive demo in JavaScript

Chapter 7

Maximal independent set

This article is about the combinatorial aspects of maximal independent sets of vertices in a graph. For other aspects of independent vertex sets in graph theory, see Independent set (graph theory). For other kinds of independent sets, see Independent set (disambiguation).

In graph theory, a **maximal independent set** (MIS) or **maximal stable set** is an independent set that is not a subset



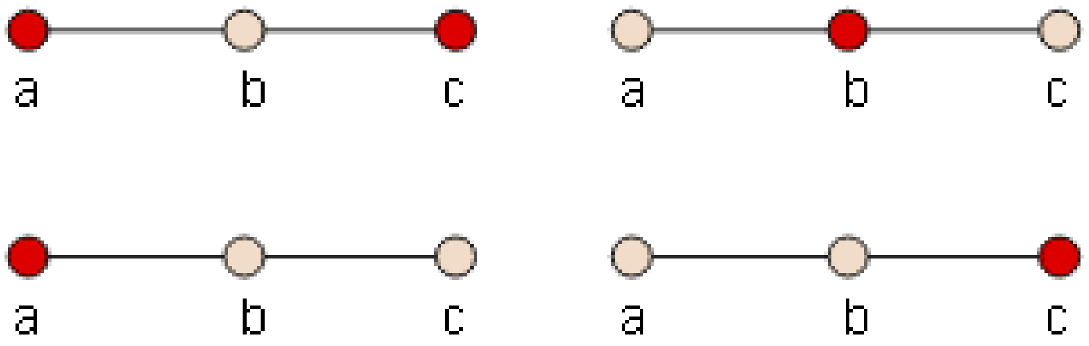
The graph of the cube has six different maximal independent sets, shown as the red vertices.

of any other independent set. In other words, there is no vertex outside the independent set that may join it because it is maximal with respect to the independent set property.

For example, in the graph P_3 , a path with three vertices a , b , and c , and two edges ab and bc , the sets $\{b\}$ and $\{a, c\}$ are both maximally independent. The set $\{a\}$ is independent, but is not maximal independent, because it is a subset of the larger independent set $\{a, c\}$. In this same graph, the maximal cliques are the sets $\{a, b\}$ and $\{b, c\}$.

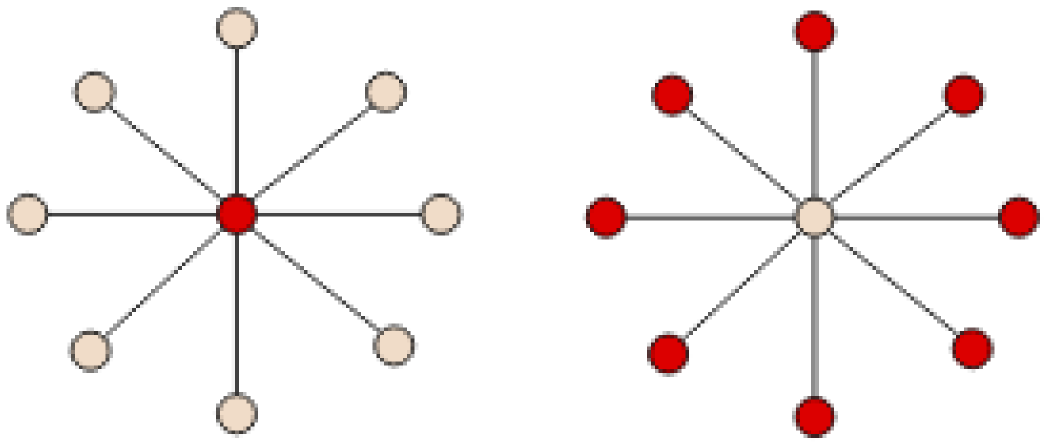
A MIS is also a dominating set in the graph, and every dominating set that is independent must be maximal independent, so MISs are also called **independent dominating sets**.

A graph may have many MISs of widely varying sizes;[1] the largest, or possibly several equally large, MISs of a graph is called a maximum independent set. The graphs in which all maximal independent sets have the same size are called well-covered graphs.



The top two P_3 graphs are maximal independent sets while the bottom two are independent sets, but not maximal. The maximum independent set is represented by the top left.

The phrase “maximal independent set” is also used to describe maximal subsets of independent elements in mathematical structures other than graphs, and in particular in vector spaces and matroids.



Two independent sets for the star graph S_8 show how vastly different in size two maximal independent sets (the right being maximum) can be.

Two algorithmic problems are associated with MISs: finding a *single* MIS in a given graph and listing *all* MISs in a given graph.

7.1 Definition

For a graph $G = (V, E)$, an independent set S is a **maximal independent set** if for $v \in V$, one of the following is true:[2]

1. $v \in S$
2. $N(v) \cap S \neq \emptyset$ where $N(v)$ denotes the neighbors of v

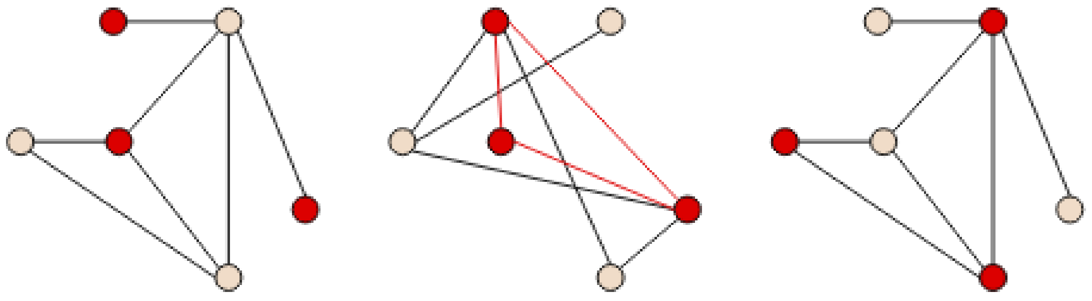
The above can be restated as a vertex either belongs to the independent set or has at least one neighbor vertex that belongs to the independent set. As a result, every edge of the graph has at least one endpoint not in S . However, it is not true that every edge of the graph has at least one, or even one endpoint in S .

Notice that any neighbor to a vertex in the independent set S cannot be in S because these vertices are disjoint by the independent set definition.

7.2 Related vertex sets

If S is a maximal independent set in some graph, it is a **maximal clique** or **maximal complete subgraph** in the complementary graph. A maximal clique is a set of vertices that induces a complete subgraph, and that is not a subset of the vertices of any larger complete subgraph. That is, it is a set S such that every pair of vertices in S is connected by an edge and every vertex not in S is missing an edge to at least one vertex in S. A graph may have many maximal cliques, of varying sizes; finding the largest of these is the maximum clique problem.

Some authors include maximality as part of the definition of a clique, and refer to maximal cliques simply as cliques.



Left is a maximal independent set. Middle is a clique, K_3 , on the graph complement. Right is a vertex cover on the maximal independent set complement.

The complement of a maximal independent set, that is, the set of vertices not belonging to the independent set, forms a **minimal vertex cover**. That is, the complement is a vertex cover, a set of vertices that includes at least one endpoint of each edge, and is minimal in the sense that none of its vertices can be removed while preserving the property that it is a cover. Minimal vertex covers have been studied in statistical mechanics in connection with the hard-sphere lattice gas model, a mathematical abstraction of fluid-solid state transitions.[3]

Every maximal independent set is a dominating set, a set of vertices such that every vertex in the graph either belongs to the set or is adjacent to the set. A set of vertices is a maximal independent set if and only if it is an independent dominating set.

7.3 Graph family characterizations

Certain graph families have also been characterized in terms of their maximal cliques or maximal independent sets. Examples include the maximal-clique irreducible and hereditary maximal-clique irreducible graphs. A graph is said to be *maximal-clique irreducible* if every maximal clique has an edge that belongs to no other maximal clique, and *hereditary maximal-clique irreducible* if the same property is true for every induced subgraph.[4] Hereditary maximal-clique irreducible graphs include triangle-free graphs, bipartite graphs, and interval graphs.

Cographs can be characterized as graphs in which every maximal clique intersects every maximal independent set, and in which the same property is true in all induced subgraphs.

7.4 Bounding the number of sets

Moon & Moser (1965) showed that any graph with n vertices has at most $3^{n/3}$ maximal cliques. Complementarily, any graph with n vertices also has at most $3^{n/3}$ maximal independent sets. A graph with exactly $3^{n/3}$ maximal independent sets is easy to construct: simply take the disjoint union of $n/3$ triangle graphs. Any maximal independent set in this graph is formed by choosing one vertex from each triangle. The complementary graph, with exactly $3^{n/3}$ maximal cliques, is a special type of Turán graph; because of their connection with Moon and Moser's bound, these graphs are also sometimes called Moon-Moser graphs. Tighter bounds are possible if one limits the size of the maximal independent sets: the number of maximal independent sets of size k in any n-vertex graph is at most

$$\lfloor n/k \rfloor^{k-(n \bmod k)} \lfloor n/k+1 \rfloor^{n \bmod k}.$$

The graphs achieving this bound are again Turán graphs.[5]

Certain families of graphs may, however, have much more restrictive bounds on the numbers of maximal independent sets or maximal cliques. If all n-vertex graphs in a family of graphs have O(n) edges, and if every subgraph of a graph in the family also belongs to the family, then each graph in the family can have at most O(n) maximal cliques, all of which have size O(1).[6] For instance, these conditions are true for the planar graphs: every n-vertex planar graph has at most $3n - 6$ edges, and a subgraph of a planar graph is always planar, from which it follows that each planar graph has O(n) maximal cliques (of size at most four). Interval graphs and chordal graphs also have at most n maximal cliques, even though they are not always sparse graphs.

The number of maximal independent sets in n-vertex cycle graphs is given by the Perrin numbers, and the number of maximal independent sets in n-vertex path graphs is given by the Padovan sequence.[7] Therefore, both numbers are proportional to powers of 1.324718, the plastic number.

7.5 Finding a single maximal independent set

7.5.1 Sequential algorithm

Given a graph G(V,E), it is easy to find a single MIS using the following algorithm:

1. Initialize I to an empty set.
2. While V is not empty:
   - Choose a node v∈V;
   - Add v to the set I;
   - Remove from V the node v and all its neighbours.
3. Return I.

The runtime is O(n) since in the worst case we have to check all nodes.

O(n) is obviously the best possible runtime for a serial algorithm. But a parallel algorithm can solve the problem much faster.

7.5.2 Random-selection parallel algorithm

The following algorithm finds a MIS in time O(log n).[2][8][9]

1. Initialize I to an empty set.
2. While V is not empty:
   - Choose a random set of vertices S ⊆ V, by selecting each vertex v independently with probability 1/(2d(v)), where d is the degree of v (the number of neighbours of v).
   - For every edge in E, if both its endpoints are in the random set S, then remove from S the endpoint whose degree is lower (i.e. has fewer neighbours). Break ties arbitrarily, e.g. using a lexicographic order on the vertex names.
   - Add the set S to I.
   - Remove from V the set S and all the neighbours of nodes in S.
3. Return I.

ANALYSIS: For each node v, divide its neighbours to *lower neighbours* (whose degree is lower than the degree of v) and *higher neighbours* (whose degree is higher than the degree of v), breaking ties as in the algorithm.

Call a node v *bad* if more than 2/3 of its neighbors are higher neighbours. Call an edge *bad* if both its endpoints are bad; otherwise the edge is *good*.

- At least 1/2 of all edges are always good. PROOF: Build a directed version of G by directing each edge to the node with the higher degree (breaking ties arbitrarily). So for every bad node, the number of out-going edges is more than 2 times the number of in-coming edges. So every bad edge, that enters a node v, can be matched to a distinct set of two edges that exit the node v. Hence the total number of edges is at least 2 times the number of bad edges.
- For every good node u, the probability that a neighbour of u is selected to S is at least a certain positive constant. PROOF: the probability that NO neighbour of u is selected to S is at most the probability that none of u's *lower neighbours* is selected. For each lower-neighbour v, the probability that it is not selected is (1-1/2d(v)), which is at most (1-1/2d(u)) (since d(u)≤d(v)). The number of such neighours is at least d(u)/3, since u is good. Hence the probability that no lower-neighbour is selected is at most 1-exp(-1/6).
- For every node u that is selected to S, the probability that u will be removed from S is at most 1/2. PROOF: This probability is at most the probability that a higher-neighbour of u is selected to S. For each higher-neighbour v, the probability that it is selected is at most 1/2d(v), which is at most 1/2d(u) (since d(v)≤d(u)). By union bound, the probability that no higher-neighbour is selected is at most d(u)/2d(u) = 1/2.
- Hence, for every good node u, the probability that a neighbour of u is selected to S and remains in S is a certain positive constant. Hence, the probability that u is removed, in each step, is at least a positive constant.
- Hence, for every good edge e, the probability that e is removed, in each step, is at least a positive constant. So the number of good edges drops by at least a constant factor each step.
- Since at least half the edges are good, the total number of edges also drops by a constant factor each step.
- Hence, the number of steps is O(log *m*), where *m* is the number of edges. This is bounded by $O(\log(n))$.

A worst-case graph, in which the average number of steps is $\Theta(\log(n))$, is a graph made of *n*/2 connected components, each with 2 nodes. The degree of all nodes is 1, so each node is selected with probability 1/2, and with probability 1/4 both nodes in a component are not chosen. Hence, the number of nodes drops by a factor of 4 each step, and the expected number of steps is $\log_4(n)$.

7.5.3 Random-priority parallel algorithm

The following algorithm is better than the previous one in that at least one new node is always added in each connected component:[10][9]

1. Initialize I to an empty set.
2. While V is not empty, each node v does the following:
   - Selects a random number r(v) in [0,1] and sends it to its neighbours;
   - If r(v) is smaller than the numbers of all neighbours of v, then v inserts itself into I, removes itself from V and tells its neighbours about this;
   - If v heard that one of its neighbours got into I, then v removes itself from V.
3. Return I.

Note that in every step, the node with the smallest number in each connected component always enters I, so there is always some progress. In particular, in the worst-case of the previous algorithm (*n*/2 connected components with 2 nodes each), a MIS will be found in a single step.

ANALYSIS:

- A node v has probability at least $\frac{1}{d(v)+d(w)}$ of being removed. PROOF: For each edge connecting a pair of nodes (v, w) , replace it with two directed edges, one from (v, w) and the other (w, v) . Notice that $|E|$ is now twice as large. For every pair of directed edges (v, w) , define two events: $(v \to w)$ and $(w \to v)$, v pre-emptively removes w and w pre-emptively removes v , respectively. The event $(v \to w)$ occurs when $r(v) < r(w)$ and $r(v) < r(x)$, where w is a neighbor of v and x is neighbor w . Recall that each node is given a random number in the same [0, 1] range. In a simple example with two disjoint nodes, each has probability

$\frac{1}{2}$ of being smallest. If there are three disjoint nodes, each has probability $\frac{1}{3}$ of being smallest. In the case of v , it has probability at least $\frac{1}{d(v)+d(w)}$ of being smallest because it is possible that a neighbor of v is also the neighbor of w , so a node becomes double counted. Using the same logic, the event $(w \rightarrow v)$ also has probability at least $\frac{1}{d(w)+d(v)}$ of being removed.

- When the events $(v \rightarrow w)$ and $(w \rightarrow v)$ occur, they remove $d(w)$ and $d(v)$ directed outgoing edges, respectively. PROOF: In the event $(v \rightarrow w)$, when v is removed, all neighboring nodes w are also removed. The number of outgoing directed edges from w removed is $d(w)$. With the same logic, $(w \rightarrow v)$ removes $d(v)$ directed outgoing edges.
- In each iteration of step 2, in expectation, half the edges are removed. PROOF: If the event $(v \rightarrow w)$ happens then all neighbours of w are removed; hence the expected number of edges removed due to this event is at least $\frac{d(w)}{d(v)+d(w)}$. The same is true for the reverse event $(w \rightarrow v)$, i.e. the expected number of edges removed is at least $\frac{d(v)}{d(w)+d(v)}$. Hence, for every undirected edge (w, v) , the expected number of edges removed due to one of these nodes having smallest value is $\frac{d(w)+d(v)}{d(w)+d(v)} = 1$. Summing over all edges, $\sum_{v,w \in E} 1 = |E|$, gives an expected number of $|E|$ edges removed every step, but each edge is counted twice (once per direction), giving $\frac{|E|}{2}$ edges removed in expectation every step.
- Hence, the expected run time of the algorithm is $3 \log_{4/3}(m) + 1$ which is $O(\log(n))$.[9]

7.5.4 Random-permutation parallel algorithm

Instead of randomizing in each step, it is possible to randomize once, at the beginning of the algorithm, by fixing a random ordering on the nodes. Given this fixed ordering, the following parallel algorithm achieves exactly the same MIS as the #Sequential algorithm (i.e. the result is deterministic):[11]

```
1. Initialize I to an empty set.
2. While V is not empty:
    • Let W be the set of vertices in V with no earlier neighbours (based on the fixed ordering);
    • Add W to I;
    • Remove from V the nodes in the set W and all their neighbours.
3. Return I.
```

Between the totally sequential and the totally parallel algorithms, there is a continuum of algorithms that are partly sequential and partly parallel. Given a fixed ordering on the nodes and a factor δ∈(0,1], the following algorithm returns the same MIS:

```
1. Initialize I to an empty set.
2. While V is not empty:
    • Select a factor δ∈(0,1].
    • Let P be the set of δ$n$ nodes that are first in the fixed ordering.
    • Let W be a MIS on P using the totally parallel algorithm.
    • Add W to I;
    • Remove from V all the nodes in the prefix P, and all the neighbours of nodes in the set W.
3. Return I.
```

Setting δ=1/*n* gives the totally sequential algorithm; setting δ=1 gives the totally parallel algorithm.

ANALYSIS: With a proper selection of the parameter δ in the partially parallel algorithm, it is possible to guarantee that the it finishes after at most log(n) calls to the fully parallel algorithm, and the number of steps in each call is at most log(n). Hence the total run-time of the partially parallel algorithm is $O(\log^2(n))$. Hence the run-time of the fully parallel algorithm is also at most $O(\log^2(n))$. The main proof steps are:

- If, in step *i*, we select $\delta = 2^i \log(n)/D$, where *D* is the maximum degree of a node in the graph, then WHP all nodes remaining after step *i* have degree at most $D/2^i$. Thus, after log(*D*) steps, all remaining nodes have degree 0 (since *D*<*n*), and can be removed in a single step.
- If, in any step, the degree of each node is at most *d*, and we select $\delta = C \log(n)/d$ (for any constant *C*), then WHP the longest path in the directed graph determined by the fixed ordering has length $O(\log(n))$. Hence the fully parallel algorithm takes at most $O(\log(n))$ steps (since the longest path is a worst-case bound on the number of steps in that algorithm).
- Combining these two facts gives that, if we select $\delta = 2^i \log(n)/D$, then WHP the run-time of the partially parallel algorithm is $O(\log(D)\log(n)) = O(\log^2(n))$.

7.6 Listing all maximal independent sets

Further information: Clique problem § Listing all maximal cliques

An algorithm for listing all maximal independent sets or maximal cliques in a graph can be used as a subroutine for solving many NP-complete graph problems. Most obviously, the solutions to the maximum independent set problem, the maximum clique problem, and the minimum independent dominating problem must all be maximal independent sets or maximal cliques, and can be found by an algorithm that lists all maximal independent sets or maximal cliques and retains the ones with the largest or smallest size. Similarly, the minimum vertex cover can be found as the complement of one of the maximal independent sets. Lawler (1976) observed that listing maximal independent sets can also be used to find 3-colorings of graphs: a graph can be 3-colored if and only if the complement of one of its maximal independent sets is bipartite. He used this approach not only for 3-coloring but as part of a more general graph coloring algorithm, and similar approaches to graph coloring have been refined by other authors since.[12] Other more complex problems can also be modeled as finding a clique or independent set of a specific type. This motivates the algorithmic problem of listing all maximal independent sets (or equivalently, all maximal cliques) efficiently.

It is straightforward to turn a proof of Moon and Moser's $3^{n/3}$ bound on the number of maximal independent sets into an algorithm that lists all such sets in time $O(3^{n/3})$.[13] For graphs that have the largest possible number of maximal independent sets, this algorithm takes constant time per output set. However, an algorithm with this time bound can be highly inefficient for graphs with more limited numbers of independent sets. For this reason, many researchers have studied algorithms that list all maximal independent sets in polynomial time per output set.[14] The time per maximal independent set is proportional to that for matrix multiplication in dense graphs, or faster in various classes of sparse graphs.[15]

7.7 Parallelization of finding maximum independent sets

7.7.1 History

The maximal independent set problem was originally thought to be non-trivial to parallelize due to the fact that the lexicographical maximal independent set proved to be P-Complete; however, it has been shown that a deterministic parallel solution could be given by an NC^1 reduction from either the maximum set packing or the maximal matching problem or by an NC^2 reduction from the 2-satisfiability problem.[16][17] Typically, the structure of the algorithm given follows other parallel graph algorithms - that is they subdivide the graph into smaller local problems that are solvable in parallel by running an identical algorithm.

Initial research into the maximal independent set problem started on the PRAM model and has since expanded to produce results for distributed algorithms on computer clusters. The many challenges of designing distributed parallel algorithms apply in equal to the maximum independent set problem. In particular, finding an algorithm that exhibits efficient runtime and is optimal in data communication for subdividing the graph and merging the independent set.

7.7.2 Complexity class

It was shown in 1984 by Karp et al. that a deterministic parallel solution on PRAM to the maximal independent set belonged in the Nick's Class complexity zoo of NC_4 .[18] That is to say, their algorithm finds a maximal independent set in $O(log^4 n)$ using $O((n/logn)^3)$, where n is the vertex set size. In the same paper, a randomized parallel solution was also provided with a runtime of $O(log^4 n)$ using $O(n^2)$ processors. Shortly after, Luby and Alon et al. independently improved on this result, bringing the minimal independent set problem into the realm of NC_2 with an $O(log^2 n)$ runtime using $O(mn^2)$ processors, where m is the number of edges in the graph.[17][8][19] In order to show that their algorithm is in NC_2 , they initially presented a randomized algorithm that uses $O(m)$ processors but could be derandomized with an additional $O(n^2)$ processors. Today, it remains an open question as to if the minimal independent set problem is in NC_1 .

7.7.3 Communication and data exchange

Distributed maximal independent set algorithms are strongly influenced by algorithms on the PRAM model. The original work by Luby and Alon et al. has led to several distributed algorithms. [20][21][22][19] In terms of exchange of bits, these algorithms had a message size lower bound per round of $O(logn)$ bits and would require additional characteristics of the graph. For example, the size of the graph would need to be known or the maximum degree of neighboring vertices for a given vertex could be queried. In 2010, Métivier et al. reduced the required message size per round to $O(1)$, which is optimal and removed the need for any additional graph knowledge.[23]

7.8 Notes

[1] Erdős (1966) shows that the number of different sizes of MISs in an *n*-vertex graph may be as large as *n* - log *n* - O(log log *n*) and is never larger than *n* - log *n*.

[2] Luby's Algorithm, in: Lecture Notes for Randomized Algorithms, Last Updated by Eric Vigoda on February 2, 2006

[3] Weigt & Hartmann (2001).

[4] Information System on Graph Class Inclusions: maximal clique irreducible graphs and hereditary maximal clique irreducible graphs.

[5] Byskov (2003). For related earlier results see Croitoru (1979) and Eppstein (2003).

[6] Chiba & Nishizeki (1985). Chiba and Nishizeki express the condition of having O(*n*) edges equivalently, in terms of the arboricity of the graphs in the family being constant.

[7] Bisdorff & Marichal (2007); Euler (2005); Füredi (1987).

[8] Luby, M. (1986). “A Simple Parallel Algorithm for the Maximal Independent Set Problem”. *SIAM Journal on Computing*. **15** (4): 1036. doi:10.1137/0215074.

[9] “Principles of Distributed Computing (lecture 7)" (PDF). ETH Zurich. Retrieved 21 February 2015.

[10] Métivier, Y.; Robson, J. M.; Saheb-Djahromi, N.; Zemmari, A. (2010). “An optimal bit complexity randomized distributed MIS algorithm”. *Distributed Computing*. **23** (5–6): 331. doi:10.1007/s00446-010-0121-5.

[11] Blelloch, Guy; Fineman, Jeremy; Shun, Julian (2012). “Greedy Sequential Maximal Independent Set and Matching are Parallel on Average”. arXiv:1202.3205 [cs.DS]. line feed character in |title= at position 71 (help)

[12] Eppstein (2003); Byskov (2003).

[13] Eppstein (2003). For a matching bound for the widely used Bron–Kerbosch algorithm, see Tomita, Tanaka & Takahashi (2006).

[14] Bomze et al. (1999); Eppstein (2005); Jennings & Motycková (1992); Johnson, Yannakakis & Papadimitriou (1988); Lawler, Lenstra & Rinnooy Kan (1980); Liang, Dhall & Lakshmivarahan (1991); Makino & Uno (2004); Mishra & Pitt (1997); Stix (2004); Tsukiyama et al. (1977); Yu & Chen (1993).

[15] Makino & Uno (2004); Eppstein (2005).

[16] Cook, Stephen (June 1983). “An overview of computational complexity” (PDF). *Commun. ACM*.

[17] Barba, Luis (October 2012). “LITERATURE REVIEW: Parallel algorithms for the maximal independent set problem in graphs” (PDF).

[18] Karp, R.M.; Wigderson, A. (1984). “A fast parallel algorithm for the maximal independent set problem”. *Proc. 16th ACM Symposium on Theory of Computing*.

[19] Alon, Noga; Laszlo, Babai; Alon, Itai (1986). "A fast and simple randomized parallel algorithm for the maximal independent set problem". *Journal of Algorithms*.

[20] Peleg, D. (2000). "Distributed computing—A Locality-sensitive approach". *SIAM Monographs on Discrete Mathematics and Applications*.

[21] Lynch, N.A. (1996). "Distributed Algorithms". *Morgan Kaufman*.

[22] Wattenhofer, R. "Chapter 4: Maximal Independent Set" (PDF).

[23] Métivier, Y.; Robson, J. M.; Saheb-Djahromi, N.; Zemmari, A. (2010). "An optimal bit complexity randomized distributed MIS algorithm". *Distributed Computing*.

7.9 References

- Bisdorff, R.; Marichal, J.-L. (2007), *Counting non-isomorphic maximal independent sets of the* n*-cycle graph*, arXiv:math.CO/0701647.

- Bomze, I. M.; Budinich, M.; Pardalos, P. M.; Pelillo, M. (1999), "The maximum clique problem", *Handbook of Combinatorial Optimization*, **4**, Kluwer Academic Publishers, pp. 1–74, CiteSeerX: 10.1.1.48.4074.

- Byskov, J. M. (2003), "Algorithms for *k*-colouring and finding maximal independent sets", *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 456–457.

- Chiba, N.; Nishizeki, T. (1985), "Arboricity and subgraph listing algorithms", *SIAM J. on Computing*, **14** (1): 210–223, doi:10.1137/0214017.

- Croitoru, C. (1979), "On stables in graphs", *Proc. Third Coll. Operations Research*, Babeş-Bolyai University, Cluj-Napoca, Romania, pp. 55–60.

- Eppstein, D. (2003), "Small maximal independent sets and faster exact graph coloring" (PDF), *Journal of Graph Algorithms and Applications*, **7** (2): 131–140, arXiv:cs.DS/0011009, doi:10.7155/jgaa.00064.

- Eppstein, D. (2005), "All maximal independent sets and dynamic dominance for sparse graphs", *Proc. Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 451–459, arXiv:cs.DS/0407036.

- Erdős, P. (1966), "On cliques in graphs", *Israel J. Math.*, **4** (4): 233–234, doi:10.1007/BF02771637, MR 0205874.

- Euler, R. (2005), "The Fibonacci number of a grid graph and a new class of integer sequences", *Journal of Integer Sequences*, **8** (2): 05.2.6, Bibcode:2005JIntS...8...26E.

- Füredi, Z. (1987), "The number of maximal independent sets in connected graphs", *Journal of Graph Theory*, **11** (4): 463–470, doi:10.1002/jgt.3190110403.

- Jennings, E.; Motycková, L. (1992), "A distributed algorithm for finding all maximal cliques in a network graph", *Proc. First Latin American Symposium on Theoretical Informatics*, Lecture Notes in Computer Science, **583**, Springer-Verlag, pp. 281–293

- Johnson, D. S.; Yannakakis, M.; Papadimitriou, C. H. (1988), "On generating all maximal independent sets", *Information Processing Letters*, **27** (3): 119–123, doi:10.1016/0020-0190(88)90065-8.

- Lawler, E. L. (1976), "A note on the complexity of the chromatic number problem", *Information Processing Letters*, **5** (3): 66–67, doi:10.1016/0020-0190(76)90065-X.

- Lawler, E. L.; Lenstra, J. K.; Rinnooy Kan, A. H. G. (1980), "Generating all maximal independent sets: NP-hardness and polynomial time algorithms", *SIAM Journal on Computing*, **9** (3): 558–565, doi:10.1137/0209042.

- Leung, J. Y.-T. (1984), "Fast algorithms for generating all maximal independent sets of interval, circular-arc and chordal graphs", *Journal of Algorithms*, **5**: 22–35, doi:10.1016/0196-6774(84)90037-3.

- Liang, Y. D.; Dhall, S. K.; Lakshmivarahan, S. (1991), *On the problem of finding all maximum weight independent sets in interval and circular arc graphs*, pp. 465–470

- Makino, K.; Uno, T. (2004), *New algorithms for enumerating all maximal cliques*, Lecture Notes in Compute Science, **3111**, Springer-Verlag, pp. 260–272.

- Mishra, N.; Pitt, L. (1997), “Generating all maximal independent sets of bounded-degree hypergraphs”, *Proc. Tenth Conf. Computational Learning Theory*, pp. 211–217, doi:10.1145/267460.267500, ISBN 0-89791-891-6.

- Moon, J. W.; Moser, L. (1965), “On cliques in graphs”, *Israel Journal of Mathematics*, **3**: 23–28, doi:10.1007/BF02760024, MR 0182577.

- Stix, V. (2004), “Finding all maximal cliques in dynamic graphs”, *Computational Optimization Appl.*, **27** (2): 173–186, doi:10.1023/B:COAP.0000008651.28952.b6

- Tomita, E.; Tanaka, A.; Takahashi, H. (2006), “The worst-case time complexity for generating all maximal cliques and computational experiments”, *Theoretical Computer Science*, **363** (1): 28–42, doi:10.1016/j.tcs.2006.06.015.

- Tsukiyama, S.; Ide, M.; Ariyoshi, H.; Shirakawa, I. (1977), “A new algorithm for generating all the maximal independent sets”, *SIAM J. on Computing*, **6** (3): 505–517, doi:10.1137/0206036.

- Weigt, Martin; Hartmann, Alexander K. (2001), “Minimal vertex covers on finite-connectivity random graphs: A hard-sphere lattice-gas picture”, *Phys. Rev. E*, **63** (5): 056127, arXiv:cond-mat/0011446, Bibcode:2001PhRvE..63e6127W, doi:10.1103/PhysRevE.63.056127.

- Yu, C.-W.; Chen, G.-H. (1993), “Generate all maximal independent sets in permutation graphs”, *Internat. J. Comput. Math.*, **47**: 1–8, doi:10.1080/00207169308804157.

Chapter 8

Minimum-cost flow problem

The **minimum-cost flow problem (MCFP)** is an optimization and decision problem to find the cheapest possible way of sending a certain amount of flow through a flow network. A typical application of this problem involves finding the best delivery route from a factory to a warehouse where the road network has some capacity and cost associated. The minimum cost flow problem is one of the most fundamental among all flow and circulation problems because most other such problems can be cast as a minimum cost flow problem and also that it can be solved very efficiently using the network simplex algorithm.

8.1 Definition

A flow network is a directed graph $G = (V, E)$ with a source vertex $s \in V$ and a sink vertex $t \in V$, where each edge $(u, v) \in E$ has capacity $c(u, v) > 0$, flow $f(u, v) \geq 0$ and cost $a(u, v)$, with most minimum-cost flow algorithms supporting edges with negative costs. The cost of sending this flow along an edge (u, v) is $f(u, v) \cdot a(u, v)$. The problem requires an amount of flow d to be sent from source s to sink t .

The definition of the problem is to minimize the **total cost** of the flow over all edges:

$$\sum_{(u,v)\in E} a(u,v) \cdot f(u,v)$$

with the constraints

8.2 Relation to other problems

A variation of this problem is to find a flow which is maximum, but has the lowest cost among the maximum flow solutions. This could be called a minimum-cost maximum-flow problem and is useful for finding minimum cost maximum matchings.

With some solutions, finding the minimum cost maximum flow instead is straightforward. If not, one can find the maximum flow by performing a binary search on d .

A related problem is the minimum cost circulation problem, which can be used for solving minimum cost flow. This is achieved by setting the lower bound on all edges to zero, and then making an extra edge from the sink t to the source s , with capacity $c(t, s) = d$ and lower bound $l(t, s) = d$, forcing the total flow from s to t to also be d .

The problem can be specialized into two other problems:

- if the capacity constraint is removed, the problem is reduced to the shortest path problem,
- if the costs are all set equal to zero, the problem is reduced to the maximum flow problem.

8.3 Solutions

The minimum cost flow problem can be solved by linear programming, since we optimize a linear function, and all constraints are linear.

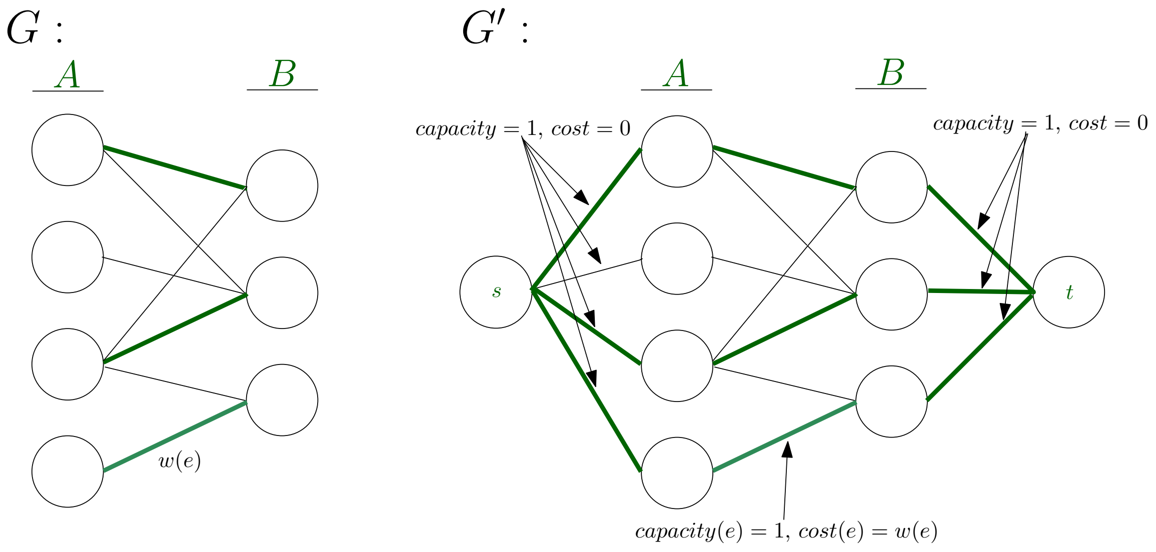
Apart from that, many combinatorial algorithms exist, for a comprehensive survey, see . Some of them are generalizations of maximum flow algorithms, others use entirely different approaches.

Well-known fundamental algorithms (they have many variations):

- *Cycle canceling*: a general primal method.
- *Minimum mean cycle canceling*: a simple strongly polynomial algorithm.
- *Successive shortest path* and *capacity scaling*: dual methods, which can be viewed as the generalizations of the Ford–Fulkerson algorithm.
- *Cost scaling*: a primal-dual approach, which can be viewed as the generalization of the push-relabel algorithm.
- *Network simplex algorithm*: a specialized version of the linear programming simplex method.
- *Out-of-kilter algorithm* by D. R. Fulkerson

8.4 Application

8.4.1 Minimum weight bipartite matching


G :
A
B
$w(e)$
G' :
A
B
$capacity = 1,\ cost = 0$
$capacity = 1,\ cost = 0$
s
t
$capacity(e) = 1,\ cost(e) = w(e)$

Reducing Minimum weight bipartite matching to minimum cost max flow problem

Given a bipartite graph $G = (A \cup B, E)$, the goal is to find the maximum cardinality matching in G that has minimum cost. Let $w\colon E \to R$ be a weight function on the edges of E. The minimum weight bipartite matching problem or assignment problem is to find a perfect matching $M \subseteq E$ whose total weight is minimized. The idea is to reduce this problem to a network flow problem.

Let $G' = (V' = A \cup B, E' = E)$. Assign the capacity of all the edges in E' to 1. Add a source vertex s and connect it to all the vertices in A' and add a sink vertex t and connect all vertices inside group B' to this vertex. The capacity of all the new edges is 1 and their costs is 0. It is proved that there is minimum weight perfect bipartite matching in G if and only if there a minimum cost flow in G'.

8.5 See also

- LEMON (C++ library)
- GNU Linear Programming Kit

8.6 References

1. ^ Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin (1993). *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc. ISBN 0-13-617549-X.
2. ^ Morton Klein (1967). "A primal method for minimal cost flows with applications to the assignment and transportation problems". *Management Science*. **14**: 205–220. doi:10.1287/mnsc.14.3.205.
3. ^ Andrew V. Goldberg and Robert E. Tarjan (1989). "Finding minimum-cost circulations by canceling negative cycles". *Journal of the ACM*. **36** (4): 873–886. doi:10.1145/76359.76368.
4. ^ Jack Edmonds and Richard M. Karp (1972). "Theoretical improvements in algorithmic efficiency for network flow problems". *Journal of the ACM*. **19** (2): 248–264. doi:10.1145/321694.321699.
5. ^ Andrew V. Goldberg and Robert E. Tarjan (1990). "Finding minimum-cost circulations by successive approximation". *Math. Oper. Res.* **15** (3): 430–466. doi:10.1287/moor.15.3.430.
6. ^ James B. Orlin (1997). "A polynomial time primal network simplex algorithm for minimum cost flows". *Mathematical Programming*. **78**: 109–129. doi:10.1007/bf02614365.
7. ^ Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin (1993). *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc. ISBN 0-13-617549-X.

8.7 External links

- LEMON C++ library with several maximum flow and minimum cost circulation algorithms

Chapter 9

Max-flow min-cut theorem

In optimization theory, the **max-flow min-cut theorem** states that in a flow network, the maximum amount of flow passing from the *source* to the *sink* is equal to the total weight of the edges in the minimum cut, i.e. the smallest total weight of the edges which if removed would disconnect the source from the sink.

The **max-flow min-cut theorem** is a special case of the duality theorem for linear programs and can be used to derive Menger's theorem and the König–Egerváry theorem.

9.1 Definitions and statement

Let $N = (V, E)$ be a network (directed graph) with s and $t \in V$ being the source and the sink of N respectively.

9.1.1 Maximum flow

Definition. The **capacity** of an edge is a mapping $c : E \to \mathbf{R}^+$, denoted by c_{uv} or $c(u, v)$. It represents the maximum amount of flow that can pass through an edge.

Definition. A **flow** is a mapping $f : E \to \mathbf{R}^+$, denoted by f_{uv} or $f(u, v)$, subject to the following two constraints:

1. Capacity Constraint:

   $\forall (u,v) \in E : \qquad f_{uv} \le c_{uv}$

2. Conservation of Flows:

   $\forall v \in V \setminus \{s,t\} : \qquad \sum_{\{u:(u,v)\in E\}} f_{uv} = \sum_{\{u:(v,u)\in E\}} f_{vu}.$

Definition. The **value of flow** is defined by

$$|f| = \sum_{v\in V} f_{sv},$$

where s is the source of N. It represents the amount of flow passing from the source to the sink.

Maximum Flow Problem. Maximize $|f|$, that is, to route as much flow as possible from s to t.

9.1.2 Minimum cut

Definition. An **s-t cut** $C = (S, T)$ is a partition of V such that $s \in S$ and $t \in T$. The **cut-set** of C is the set

$$\{(u,v) \in E \ : \ u \in S, v \in T\}.$$

Note that if the edges in the cut-set of C are removed, $|f| = 0$.

Definition. The **capacity** of an *s-t cut* is defined by

$$c(S,T) = \sum_{(u,v)\in(S\times T)\cap E} c_{uv} = \sum_{(i,j)\in E} c_{ij}d_{ij},$$

where $d_{ij} = 1$ if $i \in S$ and $j \in T$, 0 otherwise.

Minimum s-t Cut Problem. Minimize $c(S, T)$, that is, to determine S and T such that the capacity of the *S-T cut* is minimal.

9.1.3 Statement

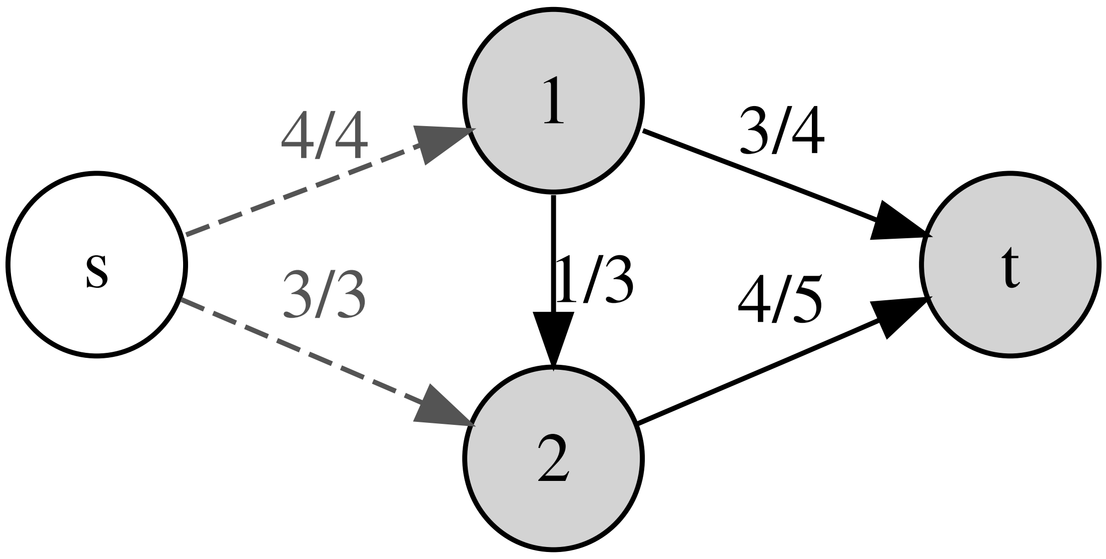
Max-flow min-cut theorem. The maximum value of an s-t flow is equal to the minimum capacity over all s-t cuts.

9.2 Linear program formulation

The max-flow problem and min-cut problem can be formulated as two primal-dual linear programs.

Note that for the given s-t cut $C = (S,T)$ if $i \in S$ then $p_i = 1$ and 0 otherwise. Therefore, p_s should be 1 and p_t should be zero. The equality in the **max-flow min-cut theorem** follows from the strong duality theorem in linear programming, which states that if the primal program has an optimal solution, x^*, then the dual program also has an optimal solution, y^*, such that the optimal values formed by the two solutions are equal.

9.3 Example



A network with the value of flow equal to the capacity of an s-t cut

The figure on the right is a network having a value of flow of 7. The vertex in white and the vertices in grey form the subsets S and T of an s-t cut, whose cut-set contains the dashed edges. Since the capacity of the s-t cut is 7, which equals to the value of flow, the max-flow min-cut theorem tells us that the value of flow and the capacity of the s-t cut are both optimal in this network.

9.4 Application

9.4.1 Generalized max-flow min-cut theorem

In addition to edge capacity, consider there is capacity at each vertex, that is, a mapping $c: V \to \mathbf{R}^+$, denoted by $c(v)$, such that the flow f has to satisfy not only the capacity constraint and the conservation of flows, but also the vertex capacity constraint

$$\forall v \in V \setminus \{s,t\}: \qquad \sum\nolimits_{\{i \in V | (iv) \in E\}} f_{iv} \le c(v).$$

In other words, the amount of *flow* passing through a vertex cannot exceed its capacity. Define an *s-t cut* to be the set of vertices and edges such that for any path from *s* to *t*, the path contains a member of the cut. In this case, the *capacity of the cut* is the sum the capacity of each edge and vertex in it.

In this new definition, the **generalized max-flow min-cut theorem** states that the maximum value of an s-t flow is equal to the minimum capacity of an s-t cut in the new sense.

9.4.2 Menger's theorem

See also: Menger's Theorem

In the undirected edge-disjoint paths problem, we are given an undirected graph $G = (V, E)$ and two vertices s and t, and we have to find the maximum number of edge-disjoint s-t paths in G.

The **Menger's theorem** states that the maximum number of edge-disjoint s-t paths in an undirected graph is equal to the minimum number of edges in an s-t cut-set.

9.4.3 Project selection problem

See also: Maximum flow problem
In the project selection problem, there are n projects and m equipments. Each project p_i yields revenue *r*(*pi*) and each equipment q_j costs *c*(*qj*) to purchase. Each project requires a number of equipments and each equipment can be shared by several projects. The problem is to determine which projects and equipments should be selected and purchased respectively, so that the profit is maximized.

Let P be the set of projects *not* selected and Q be the set of equipments purchased, then the problem can be formulated as,

$$\max\{g\} = \sum_i r(p_i) - \sum_{p_i \in P} r(p_i) - \sum_{q_j \in Q} c(q_j).$$

Since the first term does not depend on the choice of P and Q, this maximization problem can be formulated as a minimization problem instead, that is,

$$\min\{g'\} = \sum_{p_i \in P} r(p_i) + \sum_{q_j \in Q} c(q_j).$$

The above minimization problem can then be formulated as a minimum-cut problem by constructing a network, where the source is connected to the projects with capacity *r*(*pi*), and the sink is connected by the equipments with capacity *c*(*qj*). An edge (*pi*, *qj*) with *infinite* capacity is added if project p_i requires equipment q_j. The s-t cut-set represents the projects and equipments in P and Q respectively. By the max-flow min-cut theorem, one can solve the problem as a maximum flow problem.

The figure on the right gives a network formulation of the following project selection problem:

The minimum capacity of a s-t cut is 250 and the sum of the revenue of each project is 450; therefore the maximum profit *g* is 450 − 250 = 200, by selecting projects p_2 and p_3.

The idea here is to 'flow' the project profits through the 'pipes' of the equipment. If we cannot fill the pipe, the equipment's return is less than its cost, and the min cut algorithm will find it cheaper to cut the project's profit edge instead of the equipment's cost edge.

9.4.4 Image segmentation problem

See also: Maximum flow problem
In the image segmentation problem, there are n pixels. Each pixel i can be assigned a foreground value f_i or a background value b_i. There is a penalty of p_{ij} if pixels i, j are adjacent and have different assignments. The problem is to assign pixels to foreground or background such that the sum of their values minus the penalties is maximum.

Let P be the set of pixels assigned to foreground and Q be the set of points assigned to background, then the problem can be formulated as,

$$\max\{g\} = \sum_{i\in P} f_i + \sum_{i\in Q} b_i - \sum_{i\in P, j\in Q \vee j\in P, i\in Q} p_{ij}.$$

This maximization problem can be formulated as a minimization problem instead, that is,

$$\min\{g'\} = \sum_{i\in P, j\in Q \vee j\in P, i\in Q} p_{ij}.$$

The above minimization problem can be formulated as a minimum-cut problem by constructing a network where the source (orange node) is connected to all the pixels with capacity f_i, and the sink (purple node) is connected by all the pixels with capacity b_i. Two edges (i, j) and (j, i) with p_{ij} capacity are added between two adjacent pixels. The s-t cut-set then represents the pixels assigned to the foreground in P and pixels assigned to background in Q.

9.5 History

The **max-flow min-cut theorem** was proven by P. Elias, A. Feinstein, and C.E. Shannon in 1956, and independently also by L.R. Ford, Jr. and D.R. Fulkerson in the same year.

9.6 Proof

Let $G = (V, E)$ be a network (directed graph) with s and t being the source and the sink of G respectively.

Consider the flow f computed for G by Ford–Fulkerson algorithm. In the residual graph (Gf) obtained for G (after the final flow assignment by Ford–Fulkerson algorithm), define two subsets of vertices as follows:

1. A: the set of vertices reachable from s in G_f
2. A^c: the set of remaining vertices i.e. V – A

Claim. value(f) = $c(A, A^c)$, where the **capacity** of an *s-t cut* is defined by

$$c(S,T) = \sum\nolimits_{(u,v)\in S\times T} c_{uv}$$

Now, we know, $value(f) = f_{out}(A) - f_{in}(A^c)$ for any subset of vertices, A. Therefore, for value(f) = $c(A, A^c)$ we need:

- All *outgoing edges* from the cut must be **fully saturated**.
- All *incoming edges* to the cut must have **zero flow**.

To prove the above claim we consider two cases:

- In G, there exists an *outgoing edge* $(x, y), x \in A, y \in A^c$ such that it is not saturated, i.e., $f(x, y) < cxy$. This implies, that there exists a **forward edge** from x to y in G_f, therefore there exists a path from s to y in G_f, which is a contradiction. Hence, any outgoing edge (x, y) is fully saturated.
- In G, there exists an *incoming edge* $(y, x), x \in A, y \in A^c$ such that it carries some non-zero flow, i.e., $f(x, y) > 0$. This implies, that there exists a **backward edge** from x to y in G_f, therefore there exists a path from s to y in G_f, which is again a contradiction. Hence, any incoming edge (x, y) must have zero flow.

Both of the above statements prove that the capacity of cut obtained in the above described manner is equal to the flow obtained in the network. Also, the flow was obtained by Ford-Fulkerson algorithm, so it is the max-flow of the network as well.

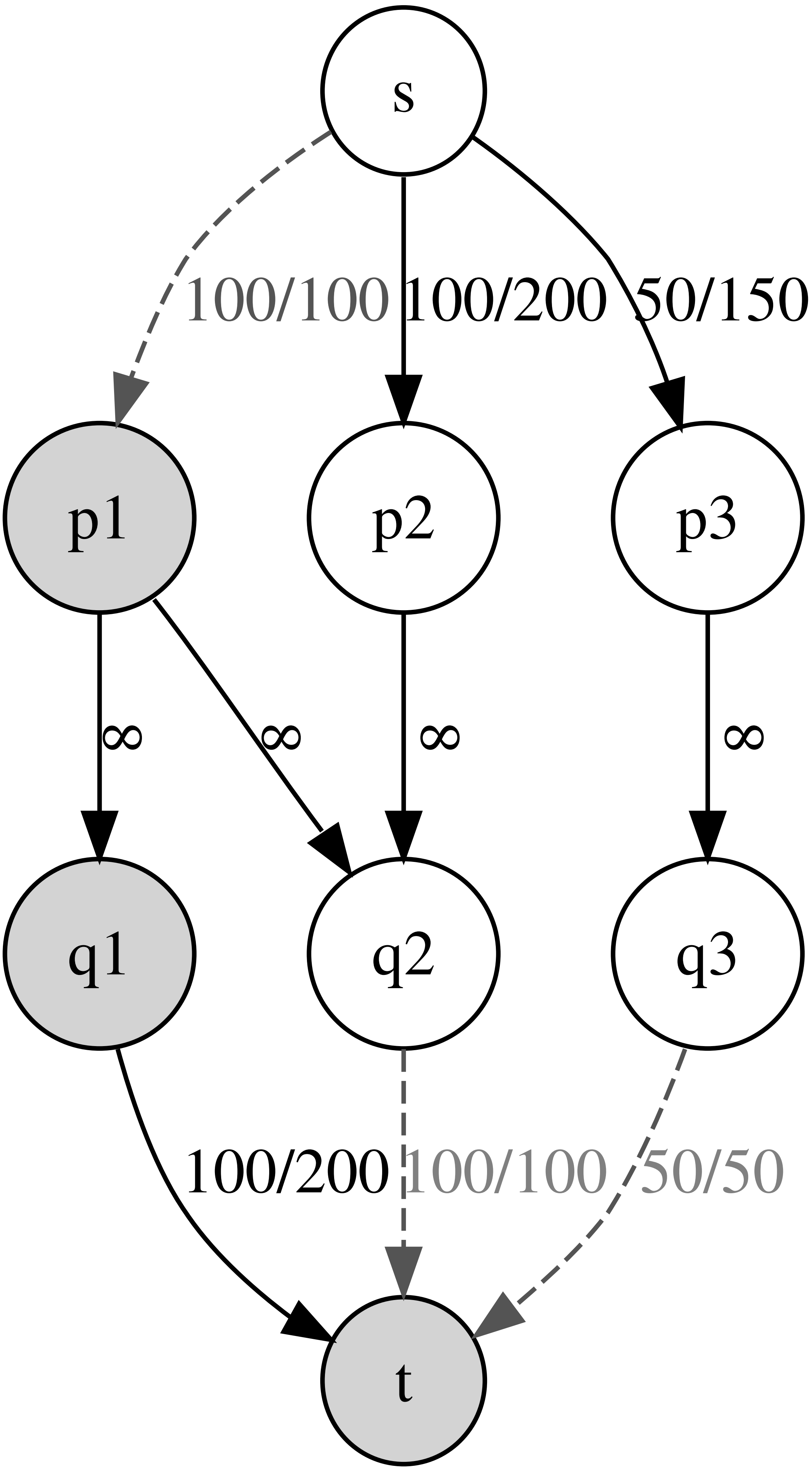
> Also, since *any flow in the network is always less than or equal to capacity of every cut possible in a network*, the above described cut is also the min-cut which obtains the max-flow.

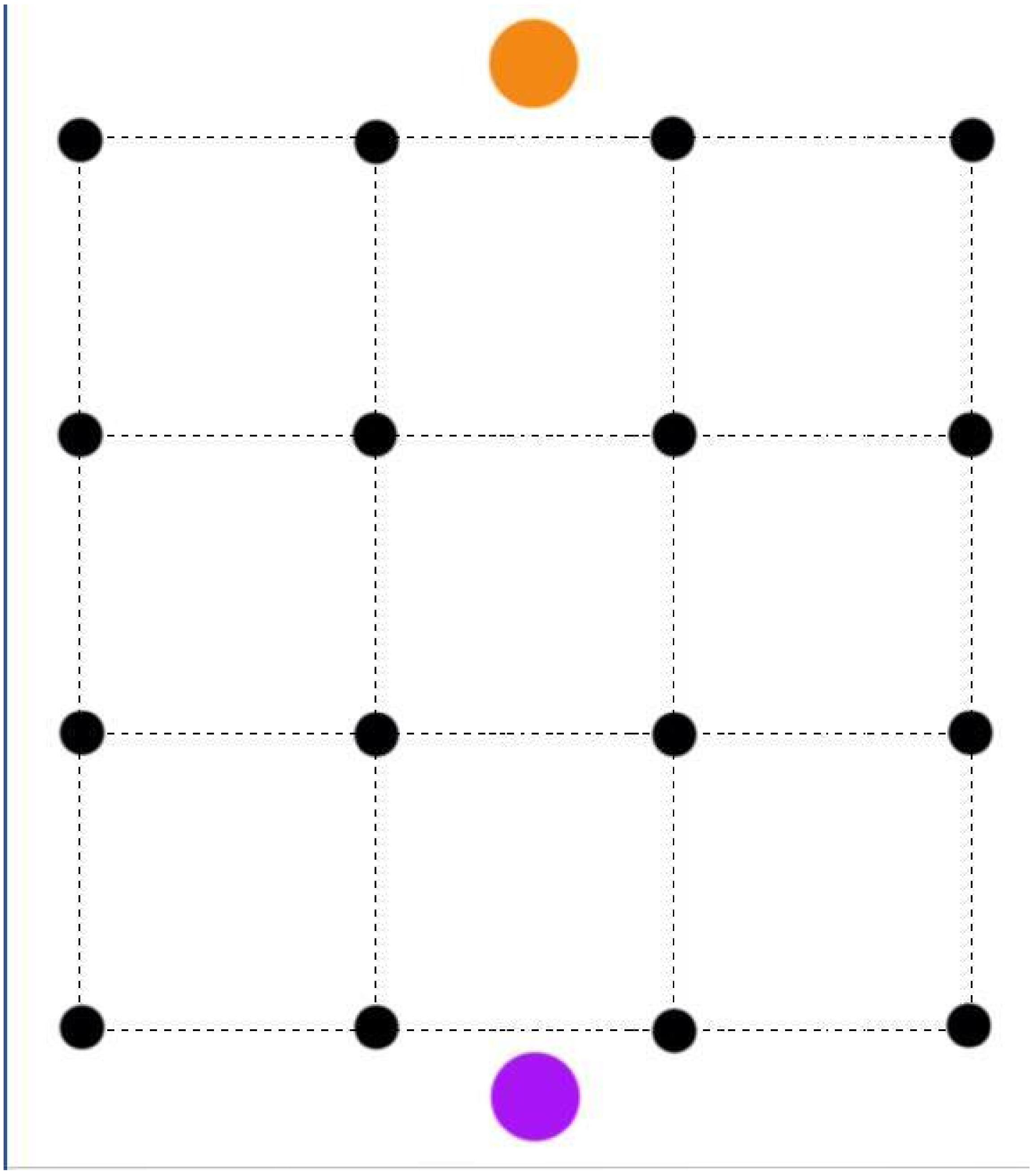
9.7 See also

- Linear programming
- Maximum flow
- Minimum cut
- Flow network
- Edmonds–Karp algorithm
- Ford–Fulkerson algorithm
- Approximate max-flow min-cut theorem

9.8 References

1. Eugene Lawler (2001). “4.5. Combinatorial Implications of Max-Flow Min-Cut Theorem, 4.6. Linear Programming Interpretation of Max-Flow Min-Cut Theorem”. *Combinatorial Optimization: Networks and Matroids*. Dover. pp. 117–120. ISBN 0-486-41453-1.
2. Christos H. Papadimitriou, Kenneth Steiglitz (1998). “6.1 The Max-Flow, Min-Cut Theorem”. *Combinatorial Optimization: Algorithms and Complexity*. Dover. pp. 120–128. ISBN 0-486-40258-4.
3. Vijay V. Vazirani (2004). “12. Introduction to LP-Duality”. *Approximation Algorithms*. Springer. pp. 93–100. ISBN 3-540-65367-8.


s
100/100
100/200
50/150
p1
p2
p3
∞
∞
∞
∞
q1
q2
q3
100/200
100/100
50/50
t



Each black node denotes a pixel.

Chapter 10

Edmonds–Karp algorithm

In computer science, the **Edmonds–Karp algorithm** is an implementation of the Ford–Fulkerson method for computing the maximum flow in a flow network in $O(V\,E^2)$ time. The algorithm was first published by Yefim (Chaim) Dinic in 1970[1] and independently published by Jack Edmonds and Richard Karp in 1972.[2] Dinic's algorithm includes additional techniques that reduce the running time to $O(V^2E)$.

10.1 Algorithm

The algorithm is identical to the Ford–Fulkerson algorithm, except that the search order when finding the augmenting path is defined. The path found must be a shortest path that has available capacity. This can be found by a breadth-first search, as we let edges have unit length. The running time of $O(V\,E^2)$ is found by showing that each augmenting path can be found in $O(E)$ time, that every time at least one of the E edges becomes saturated (an edge which has the maximum possible flow), that the distance from the saturated edge to the source along the augmenting path must be longer than last time it was saturated, and that the length is at most V. Another property of this algorithm is that the length of the shortest augmenting path increases monotonically. There is an accessible proof in *Introduction to Algorithms*.[3]

10.2 Pseudocode

For a more high level description, see Ford–Fulkerson algorithm.

algorithm EdmondsKarp **input**: C[1..n, 1..n] *(Capacity matrix)* E[1..n, 1..?] *(Neighbour lists)* s *(Source)* t *(Sink)* **output**: f *(Value of maximum flow)* F *(A matrix giving a legal flow with the maximum value)* f := 0 *(Initial flow is zero)* F := **array**(1..n, 1..n) *(Residual capacity from u to v is C[u,v] - F[u,v])* **forever** m, P := BreadthFirstSearch(C, E, s, t, F) **if** m = 0 **break** f := f + m *(Backtrack search, and write flow)* v := t **while** v ≠ s u := P[v] F[u,v] := F[u,v] + m F[v,u] := F[v,u] - m v := u **return** (f, F) **algorithm** BreadthFirstSearch **input**: C, E, s, t, F **output**: M[t] *(Capacity of path found)* P *(Parent table)* P := **array**(1..n) **for** u **in** 1..n P[u] := −1 P[s] := −2 *(make sure source is not rediscovered)* M := **array**(1..n) *(Capacity of found path to node)* M[s] := ∞ Q := queue() Q.offer(s) **while** Q.size() > 0 u := Q.poll() **for** v **in** E[u] *(If there is available capacity, and v is not seen before in search)* **if** C[u,v] - F[u,v] > 0 **and** P[v] = −1 P[v] := u M[v] := min(M[u], C[u,v] - F[u,v]) **if** v ≠ t Q.offer(v) **else return** M[t], P **return** 0, P

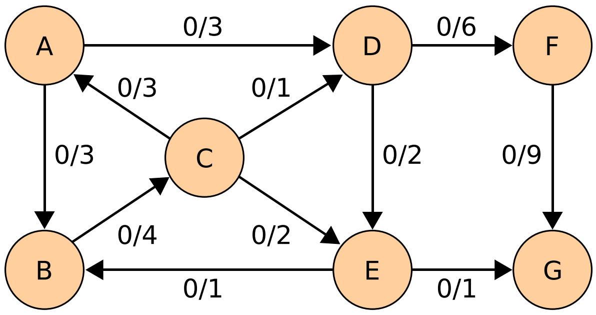
EdmondsKarp pseudo code using Adjacency nodes.

algorithm EdmondsKarp **input**: graph *(graph[v] should be the list of edges coming out of vertex v. Each edge should have a capacity, flow, source and sink as parameters, as well as a pointer to the reverse edge.)* s *(Source vertex)* t *(Sink vertex)* **output**: flow *(Value of maximum flow)* flow := 0 *(Initial flow to zero)* **repeat** *(Run a bfs to find the shortest s-t path. We use 'pred' to store the edge taken to get to each vertex, so we can recover the path afterwards)* q := **queue**() q.push(s) pred := **array**(graph.length) **while not** empty(q) cur := q.poll() **for** Edge e **in** graph[cur] **if** pred[e.t] = **null** **and** e.t ≠ s **and** e.cap > e.flow pred[e.t] := e q.push(e.t) *(Stop if we weren't able to find a path from s to t)* **if** pred[t] = null break *(Otherwise see how much flow we can send)* df := **∞** **for** (e := pred[t]; e ≠ null; e := pred[e.s]) df :=

min(df, e.cap - e.flow) *(And update edges by that amount)* **for** (e := pred[t]; e ≠ null; e := pred[e.s]) e.flow := e.flow + df e.rev.flow := e.rev.flow - df flow := flow + df **return** flow

10.3 Example

Given a network of seven nodes, source A, sink G, and capacities as shown below:



In the pairs f/c written on the edges, f is the current flow, and c is the capacity. The residual capacity from u to v is $c_f(u,v) = c(u,v) - f(u,v)$, the total capacity, minus the flow that is already used. If the net flow from u to v is negative, it *contributes* to the residual capacity.

Notice how the length of the augmenting path found by the algorithm (in red) never decreases. The paths found are the shortest possible. The flow found is equal to the capacity across the minimum cut in the graph separating the source and the sink. There is only one minimal cut in this graph, partitioning the nodes into the sets $\{A, B, C, E\}$ and $\{D, F, G\}$, with the capacity

$$c(A, D) + c(C, D) + c(E, G) = 3 + 1 + 1 = 5.$$

10.4 Notes

[1] Dinic, E. A. (1970). "Algorithm for solution of a problem of maximum flow in a network with power estimation". *Soviet Math. Doklady*. Doklady. **11**: 1277–1280.

[2] Edmonds, Jack; Karp, Richard M. (1972). "Theoretical improvements in algorithmic efficiency for network flow problems". *Journal of the ACM*. Association for Computing Machinery. **19** (2): 248–264. doi:10.1145/321694.321699.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein (2009). "26.2". *Introduction to Algorithms* (third ed.). MIT Press. pp. 727–730. ISBN 978-0-262-03384-8.

10.5 References

1. Algorithms and Complexity (see pages 63–69). http://www.cis.upenn.edu/~{ }wilf/AlgComp3.html

10.6 Text and image sources, contributors, and licenses

10.6.1 Text

- **Hopcroft–Karp algorithm** *Source:* https://en.wikipedia.org/wiki/Hopcroft%E2%80%93Karp_algorithm?oldid=739199560 *Contributors:* Edward, Twanvl, Altenmann, Oliphaunt, GregorB, Nils Grimsmo, SmackBot, Sytelus, BetacommandBot, Ruslan.Sennov, Headbomb, Mogers, David Eppstein, Mange01, Kyle the bot, Justin W Smith, Tim32, AmirOnWiki, Kathmandu guy, Addbot, Anks9b, ماني, Yobot, Nitayj, Citation bot, Michael Veksler, Gnaggnoyil, Miym, X7q, Enobrev, Exile oi, Citation bot 1, RedBot, Pazabo, WinerFresh, Thanabhat.jo, MarSukiasyan, Zhaocb, Siddhant Goel, BethNaught, Jyoteshrc, Xack~plwiki and Anonymous: 39
- **Blossom algorithm** *Source:* https://en.wikipedia.org/wiki/Blossom_algorithm?oldid=737924820 *Contributors:* Edemaine, Michael Hardy, Bkell, Giftlite, Ruud Koot, BD2412, AlanTonisson, Choess, Headbomb, A3nm, David Eppstein, Daveagp, Bender2k14, Wismon, Addbot, Favonian, Yobot, Mangarah, Philngo, Miym, Girlwithgreeneyes, יובל מדר, Citation bot 1, Markoid, Dbmikus, Dcirovic, Rezabot, Helpful Pixie Bot, YFdyh-bot, Daiyuda, Mark viking, Simonpratt, Aacombarro89 and Anonymous: 17
- **Hungarian algorithm** *Source:* https://en.wikipedia.org/wiki/Hungarian_algorithm?oldid=737400576 *Contributors:* The Anome, Fnielsen, Michael Hardy, Zweije, Giftlite, Mboverload, Macrakis, Two Bananas, Spottedowl, Andreas Kaufmann, Discospinster, 4pq1injbok, Rich Farmbrough, Bender235, LutzL, RoySmith, Shreevatsa, Bkkbrad, Ruud Koot, Miskin, Mathbot, Arichnad, Tejas81, Nils Grimsmo, Werdna, Commander Keane bot, Mcld, LoveEncounterFlow, The9muse, Watson Ladd, Klahnako, Egnever, Ivan.Savov, Jokes Free4Me, Thijs!bot, N5iln, Nkarthiks, Jjsoos, VoABot II, David Eppstein, Kope, R'n'B, VolkovBot, Slaunger, Singleheart, DrJunge, Roy hu, COBot, Justin W Smith, Vacio, Buehrenm78, Mild Bill Hiccup, Aostrander, Mike0001, Bender2k14, Mahue, Thegarybakery, Panecasareccio, XLinkBot, Purple acid, Addbot, Delaszk, Dennis.warner, Balajiivish, Delta 51, Erel Segal, Ahmadabdolkader, ArthurBot, Miym, Culipan, Ita140188, NoldorinElf, FrescoBot, Tomcatjerrymouse, $Mathe94$, Danyaljj, EmausBot, Mekeor, Hirsutism, Immunize, Dewritech, Stiel, Robert Nitsch, Pparent, Nullzero, جنگولک, Helpful Pixie Bot, Incredible Shrinking Dandy, Bob305, Chmarkine, Andyhowlett, Mslusky, NisansaDdS, Bramdj, Alexander Brock and Anonymous: 108
- **Dinic's algorithm** *Source:* https://en.wikipedia.org/wiki/Dinic'{}s_algorithm?oldid=742339936 *Contributors:* Michael Hardy, Giftlite, Andreas Kaufmann, Urod, Qwertyus, Rjwilmsi, Hashproduct, Anomalocaris, Octahedron80, Sytelus, Thijs!bot, Headbomb, JAnDbot, Magioladitis, Bbi5291, R'n'B, Mild Bill Hiccup, Sun Creator, NuclearWarfare, MystBot, Addbot, Tcshasaposse, Luckas-bot, Evergrey~enwiki, Gawi, Gilo1969, Omnipaedista, X7q, EmausBot, Dcirovic, Kasirbot, Helpful Pixie Bot, BG19bot, YFdyh-bot, Milicevic01 and Anonymous: 18
- **Stable marriage problem** *Source:* https://en.wikipedia.org/wiki/Stable_marriage_problem?oldid=742426733 *Contributors:* The Anome, BlckKnght, Michael Hardy, Wshun, Gabbe, Geoffrey~enwiki, Ronz, Nichtich~enwiki, Arthur Frayn, Evercat, Dcoetzee, Giftlite, Zarvok, Bender235, Rgdboer, Felagund, Dennis Brown, Owenjonesuk, Burn, Richwales, Oleg Alexandrov, Shreevatsa, Rjwilmsi, Ecb29, Mahahahaneapneap, Jayamohan, Teimu.tm, Cachedio, Henning Makholm, Lambiam, Mcstrother, Danrah, Cydebot, MC10, Hughdbrown, Sytelus, Not-just-yeti, Headbomb, Jojan, Widefox, Magioladitis, VoABot II, Stuart Morrow, Brusegadi, David Eppstein, User A1, Lantonov, Rponamgi, Andrewaskew, PerryTachett, Alroth, Koczy, Justin W Smith, FinnishOverlord, Plastikspork, Addbot, Dr. Universe, PV=nRT, Luckas-bot, Yobot, AnomieBOT, Erel Segal, Smk65536, Miym, Undsoweiter, FrescoBot, Darij, Codwiki, Lingust, RedBot, Trappist the monk, Vrenator, ZéroBot, Stemoc, Yukuairoy, Augurar, Dalhamir, ClueBot NG, Snotbot, Frietjes, BG19bot, Araqsya Manukyan, Rytyho usa, Timtb, Makecat-bot, E7em, Joshgans, MarketDesigner, Zieglerk, 7Sidz, Leegrc, Reddishmariposa, Nishantsny, KylimeDragon, Mtching, Jagadeesh hooli, ZatoKentai and Anonymous: 88
- **Independent set (graph theory)** *Source:* https://en.wikipedia.org/wiki/Independent_set_(graph_theory)?oldid=734815714 *Contributors:* The Anome, Twanvl, Michael Hardy, Dcoetzee, Dysprosia, MathMartin, Bkell, Giftlite, Mellum, Andris, Tyir, Manuel Anastácio, Zaslav, Syats, Spoon!, Mailer diablo, Lucienve, BD2412, Rjwilmsi, Amelio Vázquez, Marozols, Maxal, Jittat~enwiki, Chobot, Helios, YurikBot, Wavelength, RussBot, Petter Strandmark, Hv, SmackBot, Davepape, LandruBek, Ylloh, CmdrObot, Harej bot, Citrus538, Thijs!bot, Headbomb, Jeffreywarrenor, JAnDbot, David Eppstein, JoergenB, Gwern, CommonsDelinker, Rocchini, STBotD, Bramschoenmakers, Jamelan, SieBot, Flyer22 Reborn, Alexmf, Yasmar, Justin W Smith, Mahue, Arjayay, Deuler, Life of Riley, C. lorenz, MystBot, Addbot, Amirobot, Calle, AnomieBOT, Erel Segal, Citation bot, Twri, Miym, Thore Husfeldt, Freeman0119, Citation bot 1, RobinK, Trappist the monk, Aliabbas aa, Shafaet, Cjoa, Helpful Pixie Bot, BG19bot, Mikaelq, Valentina.Anitnelav, ChrisGualtieri, Adi.akbartauhidin, Bobanobahoba, AustinBuchanan, De0d0rant**Stain, Bsansouci, Frawr, Maxsd91, AnWiPaFr, Infinitestory, Nhabedi and Anonymous: 35**
- **Maximal independent set** *Source:* https://en.wikipedia.org/wiki/Maximal_independent_set?oldid=694197369 *Contributors:* Chris-martin, Silverfish, Charles Matthews, Phil Boswell, Zaslav, Rjwilmsi, Leithp, RussBot, RobertBorgersen, SmackBot, RDBury, Chris the speller, Ylloh, CBM, Hermel, Kswenson, David Eppstein, Jvimal, Austin512, Jmichael ll, Life of Riley, Addbot, DOI bot, Yobot, AnomieBOT, Erel Segal, Citation bot, Twri, Miym, Locobot, Citation bot 1, RobinK, Trappist the monk, Mentibot, Helpful Pixie Bot, Dexbot, Blucom and Anonymous: 10
- **Minimum-cost flow problem** *Source:* https://en.wikipedia.org/wiki/Minimum-cost_flow_problem?oldid=737670847 *Contributors:* Michael Hardy, Ruud Koot, Rjwilmsi, Nils Grimsmo, Mcld, Neo139, Adamarthurryan, Bwsulliv, Nczempin, Sytelus, Headbomb, David Eppstein, VolkovBot, Addbot, Rubinbot, Citation bot, Xqbot, Miym, Hs1771, Kovacspeter84, Horcrux92, RjwilmsiBot, John of Reading, Zfeinst, Ghostkeeper, Hrashi, Alexandersaschawolff, Jerry Hintze, Mark viking, Siddhant Goel, Dough34, Arash.nouri, HelpUsStopSpam, Gfabl and Anonymous: 21
- **Max-flow min-cut theorem** *Source:* https://en.wikipedia.org/wiki/Max-flow_min-cut_theorem?oldid=736885167 *Contributors:* AxelBoldt, Tomo, Michael Hardy, Chris-martin, Dan Koehl, Meekohi, Eric119, Snoyes, Dcoetzee, Dysprosia, MathMartin, Giftlite, Simon Lacoste-Julien, Paul August, ESkog, Spoon!, C S, Rshin, Finfobia, Oleg Alexandrov, LOL, Ruud Koot, Qwertyus, Kri, YurikBot, Gaius Cornelius, Petter Strandmark, Nils Grimsmo, Luc4~enwiki, Aaron Schulz, SmackBot, Mgreenbe, DHN-bot~enwiki, Mhym, Addshore, Ycl6, Kawaa, Gabn1, CmdrObot, Floridi~enwiki, Jokes Free4Me, Thijs!bot, Huttarl, JAnDbot, Douglas R. White, Stdazi, David Eppstein, Parthasarathy.kr, Aashcrahin, Ratfox, Adam Zivner, VolkovBot, Geometry guy, Jamelan, Sergio01, Aindeevar, SchreiberBike, SoxBot III, Ali Esfandiari, SteveJothen, Albambot, Addbot, Delaszk, Tcshasaposse, Matěj Grabovský, Luckas-bot, Rubinbot, TobiasdeJong, Thore Husfeldt, Yewang315, Dyno8426, יובל מדר, Louperibot, Rizzolo~enwiki, Q-lio, Darren Strash, Dinamik-bot, Horcrux92, RjwilmsiBot, EmausBot, WikitanvirBot, Zfeinst, ClueBot NG, Guof, Rezabot, Helpful Pixie Bot, BG19bot, Daiyuda, Srijanrshetty, Sharununni1, MLatinis and Anonymous: 72
- **Edmonds–Karp algorithm** *Source:* https://en.wikipedia.org/wiki/Edmonds%E2%80%93Karp_algorithm?oldid=690121168 *Contributors:* Michael Hardy, Nixdorf, Poor Yorick, Zero0000, Robbot, Chopchopwhitey, Giftlite, Martani, Sesse, Pt, Simonfl, Cburnett, RJFJR,

10.6.2 Images

10.6.3 Content license