READING PROJECT IN THE COURSE CS774 OPTIMIZATION TECHINIQUES ON PROBLEMS OF COMBINATORIAL OPTIMIZATION

BY

PRATIK MISHRA 14493

ABHINAV PRAKASH 14014

Introduction to the Travelling Salesman Problem

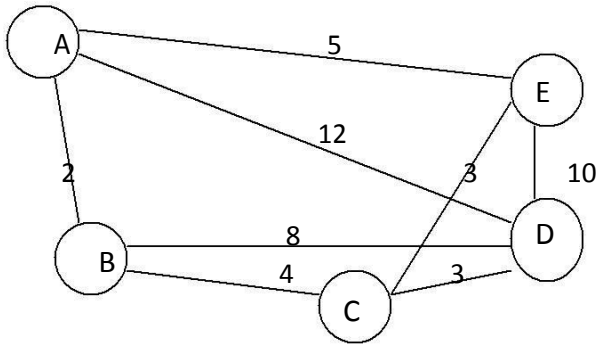
The travelling salesman problem (TSP) asks the following question: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

The traveling salesman problem can be described as follows:

TSP = {(*G, f, t*): *G* = (*V, E*) a complete graph,
f is a function $V \times V \rightarrow Z$, *t*
$\in Z$,
G is a graph that contains a traveling salesman tour with cost that does not exceed *t*}.

Example:

Consider the following set of cities:



A graph with weights on its edges.

The problem lies in finding a minimal path passing from all vertices once. For example the path Path1 {A, B, C, D, E, A} and the path Path2 {A, B, C, E, D, A} pass all the vertices but Path1 has a total length of 24 and Path2 has a total length of 31.

Definition:

A Hamiltonian cycle is a cycle in a graph passing through all the vertices once.

Theorem:
The traveling salesman problem is NP-complete.

Proof:
First, we have to prove that TSP belongs to NP. If we want to check a tour for credibility, we check that the tour contains each vertex once. Then we sum the total cost of the edges and

finally we check if the cost is minimum. This can be completed in polynomial time thus TSP belongs to NP.
Secondly we prove that TSP is NP-hard. One way to prove this is to show that Hamiltonian cycle $\leq_P$ TSP (given that the Hamiltonian cycle problem is NP-complete). Assume $G = (V, E)$ to be an instance of Hamiltonian cycle. An instance of TSP is then constructed. We create the complete graph $G' = (V, E')$, where $E' = \{(i, j): i, j \in V \text{ and } i \neq j\}$. Thus, the cost function is defined as:

$$t(i,,j) = \begin{cases} 0 \text{ if } (i, j) \in E, \\ 1 \text{ if } (i, j) \notin E. \end{cases} \qquad 1$$

Now suppose that a Hamiltonian cycle h exists in G. It is clear that the cost of each edge in h is 0 in G as each edge belongs to E. Therefore, h has a cost of 0 in G' . Thus, if graph G has a Hamiltonian cycle then graph G'has a tour of 0 cost.
Conversely, we assume that G' has a tour h' of cost at most 0. The cost of edges in E' are 0 and 1 by definition. So each edge must have a cost of 0 as the cost of h' is 0. We conclude that h' contains only edges in E.
So we have proven that G has a Hamiltonian cycle if and only if G' has a tour of cost at most 0. Thus TSP is NP-Complete.

COVERING THE BASICS

THE CLASS P

A decision problem *D* is *solvable in polynomial time* or *in the class P,* if there exists an algorithm *A* such that

- *A* takes instances of *D* as inputs.
- *A* always outputs the correct answer “Yes” or “No”.
- There exists a polynomial *p* such that the execution of *A* on inputs of size *n* always terminates in *p(n)* or fewer steps.

THE CLASS NP

A decision problem is *non deterministically polynomial-time solvable* or *in the class NP* if there exists an algorithm *A* such that

- *A* takes as inputs potential witnesses for “yes” answers to problem *D.*
- *A* correctly distinguishes true witnesses from false witnesses.
- There exists a polynomial *p* such that for each potential witnesses of each instance of size *n* of *D,* the execution of the algorithm *A* takes at most *p(n)* steps.

POLYNOMIAL TIME REDUCTION

Let *E* and *D* be two decision problems. We say that *D* is *polynomial-time reducible to E* if there exists an algorithm *A* such that

- *A* takes instances of *D* as inputs and always outputs the correct answer "Yes" or "No" for each instance of *D.*
- *A* uses as a subroutine a hypothetical algorithm *B* for solving *E*.
- There exists a polynomial *p* such that for every instance of *D* of size *n* the algorithm *A* terminates in at most *p(n)* steps *if each call of the subroutine B is counted as only m steps, where m is the size of the actual input of B.*

NP-COMPLETE AND NP-HARD

A decision problem *E* is *NP-complete* if *every* problem in the class *NP* is polynomial-time reducible to *E.*

A problem X is NP-hard, if there is an NP-complete problem Y, such that Y is

Reducible to X in polynomial time.

APPLICATIONS OF THE TRAVELLING SALESMAN PROBLEM

Much of the work on the TSP is motivated by its use as a platform for the study of general methods that can be applied to a wide range of discrete optimization problems. This is not to say, however, that the TSP does not find applications in many fields. Indeed, the numerous direct applications of the TSP bring life to the research area and help to direct future work.

The TSP naturally arises as a subproblem in many transportation and logistics applications, for example the problem of arranging school bus routes to pick up the children in a school district. This bus application is of important historical significance to the TSP, since it provided motivation for Merrill Flood, one of the pioneers of TSP research in the 1940s. A second TSP application from the 1940s involved the transportation of farming equipment from one location to another to test soil, leading to mathematical studies in Bengal by P. C. Mahalanobis and in Iowa by R. J. Jessen. More recent applications involve the scheduling of service calls at cable firms, the delivery of meals to homebound persons, the scheduling of stacker cranes in warehouses, the routing of trucks for parcel post pickup, and a host of others.

Although transportation applications are the most natural setting for the TSP, the simplicity of the model has led to many interesting applications in other areas. A classic example is the scheduling of a machine to drill holes in a circuit board or other object. In this case the holes to be drilled are the cities, and the cost of travel is the time it takes to move the drill head from one hole to the next. The

technology for drilling varies from one industry to another, but whenever the travel time of the drilling device is a significant portion of the overall manufacturing process then the TSP can play a role in reducing costs.

Methods to solve the traveling salesman problem

Since TSP is a known NP-Hard problem, it does not have any polynomial time optimum solutions. We do not consider the exponential Dynamic Programming solution here because of its infeasibility. We look at approximate solutions for TSP that achieve polynomial time.

Using the triangle inequality to solve the traveling salesman problem

Definition:
If for any set of vertices $a, b, c \in V$, it is true that t $(a, c) \leq t(a, b) + t(b, c)$ where t is the cost function, we say that t satisfies the triangle inequality.

First, we create a minimum spanning tree the weight of which is a lower bound on the cost of an optimal traveling salesman tour. Using this minimum spanning tree we will create a tour the cost of which is at most 2 times the weight of the spanning tree. We present the algorithm that performs these computations using the MST-Prim algorithm.

Approximation-TSP
Input: A complete graph $G (V, E)$
Output: A Hamiltonian cycle

1. Select a "root" vertex $r \in V [G]$.
2. Use MST-Prim (G, c, r) to compute a minimum spanning tree from r.
3. Assume L to be the sequence of vertices visited in a preorder tree walk of T.
4. Return the Hamiltonian cycle H that visits the vertices in the order L.

Theorem :
Approximation-TSP is a 2-approximation algorithm with polynomial cost for the traveling salesman problem given the triangle inequality.

Proof:
Approximation-TSP costs polynomial time as was shown before.

Assume H^* to be an optimal tour for a set of vertices. A spanning tree is constructed by deleting edges from a tour. Thus, an optimal tour has more weight than the minimum- spanning tree,

which means that the weight of the minimum spanning tree forms a lower bound on the weight of an optimal tour.

$$c(t) \leq c(H^*). \qquad 2$$

Let a full walk of T be the complete list of vertices when they are visited regardless if they are visited for the first time or not. The full walk is *W*. In our example:
W = A, B, C, B, D, B, E, B, A,.
The full walk crosses each edge exactly twice. Thus, we can write:

$$c(W) = 2c(T). \qquad 3$$

From equations 10.2 and 10.3 we can write that

$$c(W) \leq 2c(H^*), \qquad 4$$

Which means that the cost of the full path is at most 2 time worse than the cost of an optimal tour. The full path visits some of the vertices twice which means it is not a tour. We can now use the triangle inequality to erase some visits without increasing the cost. The fact we are going to use is that if a vertex a is deleted from the full path if it lies between two visits to b and c the result suggests going from b to c directly.
In our example we are left with the tour: A, B, C, D, E, A. This tour is the same as the one we get by a preorder walk. Considering this preorder walk let H be a cycle deriving from this walk. Each vertex is visited once so it is a Hamiltonian cycle. We have derived H deleting edges from the full walk so we can write:

$$c(H) \leq c(W) \qquad 5$$

From 10.4 and 10.5 we can imply:

$$c(H) \leq 2\ c(H^*). \qquad 6$$

This last inequality completes the proof.

BRANCH AND BOUND ALGORITHM

The branch and bound algorithm converts the asymmetric traveling salesman problem into an assignment problem. Consider a graph V that contains all the cities. Consider Π being the set of all the permutations of the cities, thus covering all possible solutions. Consider a permutation of this set $\pi \in \Pi$ in which each city is assigned a successor, say *i*,for the π_i city. So a tour might be (1, $\pi(1), \pi(\pi(1)),\ldots,$1). If the number of the cities in the tour is n then the permutation is called a cyclic permutation. The assignment problem tries to find such cyclic permutations and the asymmetric traveling salesman problem seeks such permutations but with the constraint that they have a minimal cost. The branch and bound algorithm firstly seeks a solution of the assignment problem. The cost to find a solution to the assignment problem for n cities is quite large and is asymptotically $O(n^3)$.
If this is a complete tour, then the algorithm has found the solution to the asymmetric traveling

salesman problem too. If not, then the problem is divided in several sub-problems. Each of these sub-problems excludes some arcs of a sub-tour, thus excluding the sub-tour itself. The way the algorithm chooses which arc to delete is called branching rules. It is very important that there are no duplicate sub-problems generated so that the total number of the sub-problems is minimized.

Carpaneto and Toth have proposed a rule that guarantees that the sub-problems are independent. They consider the included arc set and select a minimum number of arcs that do not belong to that set. They divide the problem as follows. Symbolize as E the excluded arc set and as I the included arc set. The I is to be decomposed. Let t arcs of the selected sub-tour $x_1x_2 ...x_n$ not to belong to I. The problem is divided into t children so that the j_{th} child has E_j excluded arc set and I_j included arc set. We can now write:

$$Ej = E \cup \{x_j\}$$

$$Ii - I \cup \{x_1, x_2, ..., x_{j-1}\} \quad k = 1,2,..j$$

But x_j is an excluded arc of the j_{th} sub-problem and an included arc in the $(j+1)_{st}$ problem. This means that a tour produced by the $(j+1)_{st}$ problem may have the x_j arc but a tour produced by the j_{th} problem may not contain the arc. This means that the two problems cannot generate the same tours, as they cannot contain the same arcs. This guarantees that there are no duplicate tours.

Complexity of the branch and bound algorithm

There has been a lot of controversy concerning the complexity of the branch and bound algorithm. Bellmore and Malone have stated that the algorithm runs in polynomial time. They have treated the problem as a statistical experiment assuming that the i_{th} try of the algorithm is successful if it finds a minimal tour for the I_{th} sub- problem. They assumed that the probability of the assignment problem to find the solution to the asymmetric traveling salesman problem is e/n where n is the number of the cities.

Smith concluded that under some more assumptions the complexity of the algorithm is $O(n^3 ln(n))$

The assumptions made to reach this result are too optimistic.

HEURISTIC SOLUTIONS

1. Nearest Neighbour O(N^2)

This is perhaps the simplest and most straightforward TSP heuristic. The key to this algorithm is to always visit the nearest city.

1. Select a random city.
2. Find the nearest unvisited city and go there.
3. Are there any unvisited cities left? If yes, repeat step 2.
4. Return to the first city.

The Nearest Neighbor algorithm will often keep its tours within 25% of the Held- Karp lower bound.

2. Greedy Algorithm O($N^2 * \log_2 N$)

1. Sort all edges.
2. Select the shortest edge and add it to our tour if it doesn't violate any of the constraints we saw on the last slide.
3. Do we have N edges in our tour? If no, repeat step 2.

The Greedy algorithm normally keeps within 15- 20% of the Held-Karp lower bound.

3. Nearest Insertion, O(N^2)

1. Select the shortest edge, and make a sub tour of it.
2. Select a city not in the sub tour, having the shortest distance to any one of the cities in the sub tour.
3. Find an edge in the sub tour such that the cost of inserting the selected city between the edge's cities will be minimal.
4. Repeat step 2 until no more cities remain.

4. Convex Hull O($N^2 * \log_2 N$)

1. Find the convex hull of our set of cities, and make it our initial sub tour.
2. For each city not in the sub tour, find its cheapest insertion (as in step 3 of Nearest Insertion). Then chose the city with the least cost/increase ratio, and insert it.
3. Repeat step 2 until no more cities remain.

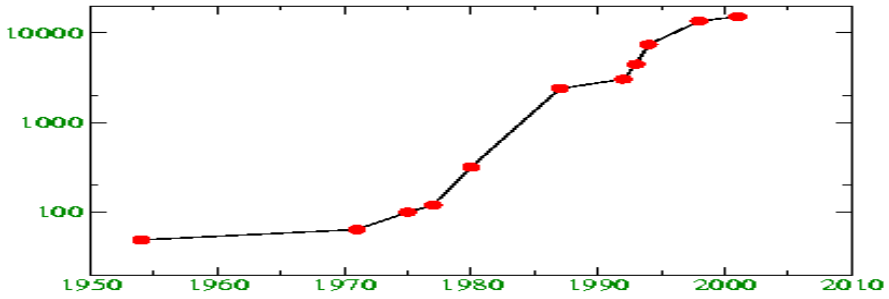
5. CHRISTOFIDES HEURISTIC

Most heuristics can only guarantee a worst-case ratio of 2 (i.e. a tour with twice the length of the optimal tour). Professor Nicos Christofides extended one of these algorithms and concluded that the worst-case ratio of that extended algorithm was 3/2.

This algorithm is commonly known as Christofides heuristic.

1. Create a minimum spanning tree *T* of *G*.
2. Let *O* be the set of vertices with odd degree in *T*. By the handshaking lemma, *O* has an even number of vertices.
3. Find a minimum-weight perfect matching *M* in the induced subgraph given by the vertices from *O*.
4. Combine the edges of *M* and *T* to form a connected multigraph *H* in which each vertex has even degree.
5. Form an Eulerian circuit in *H*.
6. Make the circuit found in previous step into a Hamiltonian circuit by skipping repeated vertices (*shortcutting*).

Milestones in the Solution of the TSP



Year	Cities	Team
1954	49	Dantzig, Fulkerson, and Johnson
1971	54	Held and Karp
1975	100	Camerini, Fratta, and Maffioli
1977	120	Groetschel
1980	318	Crowder and Padberg
1987	2,392	Padberg and Rinaldi
1992	3,038	Concorde
1993	4,461	Concorde
1994	7,397	Concorde
1998	13,509	Concorde
2001	15,112	Concorde

Factor of 126

2025: 1.9 mIIIIon?

Approximation Algorithms

Overview

Suppose we are given an **NP**-complete problem to solve. Even though（assuming $P \neq NP$）we can't hope for a polynomial-time algorithm that always gets the best solution, can we develop polynomial-time algorithms that always produce a "*pretty good*" solution? In this survey we consider such $approximation\ algorithms$, for several important problems. Specific topics in this survey include:

• 2-approximation for vertex cover via greedy matchings.

• Greedy O(log n) approximation for set-cover.

Introduction

Suppose we are given a problem for which (perhaps because it is NP-complete) we can't hope for a fast algorithm that always gets the best solution. Can we hope for a fast algorithm that guarantees to get at least a "pretty good" solution? E.g., can we guarantee to find a solution that's within 10% of optimal? If not that, then how about within a factor of 2 of optimal? Or, anything non-trivial?

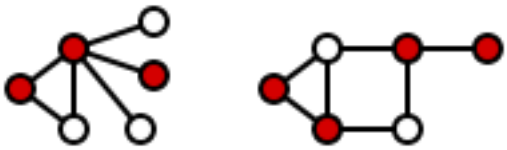
The class of NP-complete problems are all equivalent in the sense that a polynomial-time algorithm to solve any one of them would imply a polynomial-time algorithm to solve all of them (and, moreover, to solve any problem in NP). However, the difficulty of getting a good approximation to these problems varies quite a bit. In this survey we will examine several important NP-complete problems and look at to what extent we can guarantee to get approximately optimal solutions, and by what algorithms.

• Minimum Vertex Cover Problem

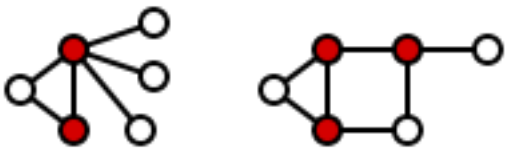
In the mathematical discipline of graph theory, a **vertex cover** (sometimes **node cover**) of a graph is a set of vertices such that each edge of the graph is incident to at least one vertex of the set. The problem of finding a **minimum vertex cover** is a classical optimization problem in computer science and is a typical example of an NP-hard optimization problem that has an approximation algorithm.
The minimum vertex cover problem can be formulated as a half-integral linear program whose dual linear program is the maximum matching problem.
Its decision version, the **vertex cover problem**, was one of Karp's 21 NP-complete problems and is therefore a classical NP-complete problem in computational complexity theory.
Furthermore, the vertex cover problem is fixed-parameter tractable and a central problem in parameterized complexity theory.

Definition

Formally, a vertex cover V' of an undirected graph $G = (V, E)$ is a subset of V such that $(u, v)\, \varepsilon\, E \quad \Rightarrow \quad u\, \varepsilon\, V' \quad or \quad v\, \varepsilon\, V'$, that is to say it is a set of vertices V' where every edge has at least one endpoint in the vertex cover V'. Such a set is said to *cover* the edges of G. The following figure shows two examples of vertex covers, with some vertex cover V' marked in red.



A *minimum vertex cover* is a vertex cover of smallest possible size. The vertex cover number τ is the size of a minimum vertex cover, i.e. $\tau = |V'|$. The following figure shows examples of minimum vertex covers in the previous graphs.



Properties

- A set of vertices is a vertex cover if and only if its complement is an independent set.
- Consequently, the number of vertices of a graph is equal to its minimum vertex cover number plus the size of a maximum independent set (Gallai 1959).

⋆ In graph theory, an **independent 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.

Computational problem

The **minimum vertex cover problem** is the optimization problem of finding a smallest vertex cover in a given graph.

INSTANCE: Graph G

OUTPUT: Smallest number k such that G has a vertex cover of size k.

If the problem is stated as a decision problem, it is called the **vertex cover problem**:

INSTANCE: Graph G and positive integer k.

QUESTION: Does G have a vertex cover of size at most k ?

The vertex cover problem is an NP-complete problem: it was one of Karp's 21 NP-complete problems. It is often used in computational complexity theory as a starting point for NP-hardness proofs.

ILP Formulation

Assume that every vertex has an associated cost of $c(v) \geq 0$. The (weighted) minimum vertex cover problem can be formulated as the following integer linear program (ILP).

minimise $\sum_{v \varepsilon V} c(v)\, x_v$		(minimise the total cost)
provided $x_u + x_v \geq 1$	$\forall\ (u,v) \varepsilon E$	(cover every edge of the graph)
$x_v \varepsilon \{0,1\}$	$\forall\ v \varepsilon V$	(every edge is either in the vertex cover or not)

This ILP belongs to the more general class of ILPs for covering problems. The integrality gap of this ILP is 2, so its relaxation gives a factor-2 approximation algorithm for the minimum vertex cover problem. Furthermore, the linear programming relaxation of that ILP is *half-integral*, that is, there exists an optimal solution for which each entry x_v is either 0, ½ or 1.

Exact evaluation

The decision variant of the vertex cover problem is NP-complete, which means it is unlikely that there is an efficient algorithm to solve it exactly. NP-completeness can be proven by reduction from 3-satisfiability or, as Karp did, by reduction from the clique problem. Vertex cover remains NP-complete even in cubic graphs and even in planar graphs of degree at most 3.
For bipartite graphs, the equivalence between vertex cover and maximum matching described by König's theorem allows the bipartite vertex cover problem to be solved in polynomial time.

For tree graphs, an algorithm finds a minimal vertex cover in polynomial time by finding the first leaf in the tree and adding its parent to the minimal vertex cover, then deleting the leaf and parent and all associated edges and continuing repeatedly until no nodes remain in the tree.

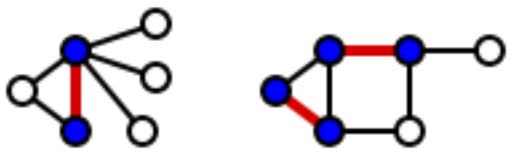
Fixed-parameter tractability

An exhaustive search algorithm can solve the problem in time $2k\,n^{O(1)}$. Vertex cover is therefore fixed-parameter tractable, and if we are only interested in small *k*, we can solve the problem in polynomial time. One algorithmic technique that works here is called *bounded search tree algorithm*, and its idea is to repeatedly choose some vertex and recursively branch, with two cases at each step: place either the current vertex or all its neighbours into the vertex cover. The algorithm for solving vertex cover that achieves the best asymptotic dependence on the parameter runs in time $O(1.27^k + k.n)$. Under reasonable complexity-theoretic assumptions, namely the exponential time hypothesis, the running time cannot be improved to $2^{O(k)}$, even when n is $O(k)$.

However, for planar graphs, and more generally, for graphs excluding some fixed graph as a minor, a vertex cover of size *k* can be found in time $2^{O(\sqrt{k})}n^{O(1)}$, the problem is subexponential fixed-parameter tractable. This algorithm is again optimal, in the sense that, under the exponential time hypothesis no algorithm can solve vertex cover on planar graphs in time $2^{o(\sqrt{k})}n^{O(1)}$.

Approximate evaluation

One can find a factor-2 approximation by repeatedly taking *both* endpoints of an edge into the vertex cover, then removing them from the graph. Put otherwise, we find a maximal matching *M* with a greedy algorithm and construct a vertex cover *C* that consists of all endpoints of the edges in *M*. In the following figure, a maximal matching *M* is marked with red, and the vertex cover *C* is marked with blue.



The set *C* constructed this way is a vertex cover: suppose that an edge *e* is not covered by *C*; then $M \cup \{e\}$ is a matching and $e \notin M$, which is a contradiction with the assumption that *M* is maximal. Furthermore, if $e = \{u, v\} \in M$, then any vertex cover – including an optimal vertex cover – must contain *u* or *v* (or both); otherwise the edge *e* is not covered. That is, an optimal cover contains at least *one* endpoint of each edge in *M*; in total, the set *C* is at most 2 times as large as the optimal vertex cover.

This simple algorithm was discovered independently by Fanica Gavril and Mihalis Yannakakis.

More involved techniques show that there are approximation algorithms with a slightly better approximation factor. For example, an approximation algorithm with an approximation factor of $2 - \Theta(\frac{1}{\sqrt{log\,|V|}})$ is known. The problem can be approximated with an approximation factor $\frac{2}{1+\delta}$ in $\delta -$ dense graphs.

Pseudo Code

```
APPROXIMATION-VERTEX-COVER(G):
C = ∅
E' = G.E

while E' ≠ ∅ :
    let (u, v) be an arbitrary edge of E'
    C = C ∪ {u, v}
    remove from E' every edge incident on either u or v

return C
```

Theorem: If the optimal solution uses k sets, the greedy algorithm finds a solution with at most $2k$ sets.

Proof :

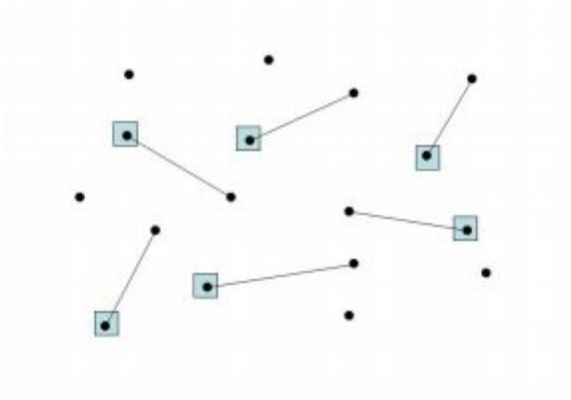


Figure 2: Another instance of Vertex Cover and its optimal cover shown in blue squares

The optimum vertex cover must cover every edge in M. So, it must include at least one of the endpoints of each edge $\in M$, where no 2 edges in M share an endpoint. Hence, optimum vertex cover must have size

$$OPT(I) \geq |M|$$

But the algorithm A return a vertex cover of size $2|M|$, so

$$\forall I \qquad A(I) = 2|M| \leq 2 \times OPT(I)$$

implying that A is a $2-approximation\ algorithm.$

Inapproximability

No better constant-factor approximation algorithm than the above one is known. The minimum vertex cover problem is APX-complete, that is, it cannot be approximated arbitrarily well unless **P** = **NP**. Using techniques from the PCP theorem, Dinur and Safra proved in 2005 that minimum vertex cover cannot be approximated within a factor of 1.3606 for any sufficiently large vertex degree unless **P** = **NP**. Moreover, if the unique games conjecture is true then minimum vertex cover cannot be approximated within any constant factor better than 2.

Although finding the minimum-size vertex cover is equivalent to finding the maximum-size independent set, as described above, the two problems are not equivalent in an approximation-preserving way: The Independent Set problem has *no* constant-factor approximation unless **P** = **NP**.

- Set Cover Cover Problem

The **set cover problem** is a classical question in combinatorics, computer science and complexity theory. It is one of Karp's 21 NP-complete problems shown to be NP-complete in 1972.

It is a problem "whose study has led to the development of fundamental techniques for the entire field" of approximation algorithms.

Given a set of elements $\{1,2,...,n\}$ and a collection S of m sets whose union equals the universe, the set cover problem is to identify the smallest sub-collection of S whose union equals the universe. For example, consider the universe $U = \{1,2,3,4,5\}\}$ and the collection of sets $S = \{\{1,2,3\},\{2,4\},\{3,4\},\{4,5\}\}$. Clearly the union of S is U. However, we can cover all of the elements with the following, smaller number of sets:

$\{\{1,2,3\},\{2,4\}\}$

More formally, given a universe Ω and a family Ψ of subsets of Ω a *cover* is a subfamily $\zeta \subseteq \Psi$ of sets whose union is Ω. In the set covering decision problem, the input is a pair (Ω, Ψ) and an integer k; the question is whether there is a set covering of size k or less. In the set covering optimization problem, the input is a pair (Ω, Ψ) ,and the task is to find a set covering that uses the fewest sets.

The decision version of set covering is NP-complete, and the optimization/search version of set cover is NP-hard .

If each set is assigned a cost, it becomes a *weighted* set cover problem.

Integer linear program formulation

The minimum set cover problem can be formulated as the following integer linear program(ILP).

minimize	$\sum_{S \in \mathcal{S}} x_S$		(minimize the number of sets)
subject to	$\sum_{S:e \in \mathcal{S}} x_S \geqslant 1$	for all $e \in \Omega$	(cover every element of the universe)
	$x_S \in \{0,1\}$	for all $S \in \Psi$	(every set is either in the set cover or not)

This ILP belongs to the more general class of ILPs for covering problems. The integrality gap of this ILP is at most $log\ n$, so its relaxation gives a factor-$log\ n$ approximation algorithm for the minimum set cover problem (where n is the size of the universe).

A Greedy Approximation Algorithm

Algorithm 1: Greedy-Set-Cover (X, F)

1 $U \leftarrow X$

2 $C \leftarrow \varnothing$

3 While $U \neq 0$

4 do select an $S \in F$ that maximizes $|S \cap U|$

5 $U \leftarrow U - S$

6 $C \leftarrow C \cup \{S\}$

7 return C

Theorem: If the optimal solution uses k sets, the greedy algorithm finds a solution with at most $k \ln n$ sets

Proof: Since the optimal solution uses k sets, there must some set that covers at least a $\frac{1}{k}$ fraction of the points. The algorithm chooses the set that covers the most points, so it covers at least that many. Therefore, after the first iteration of the algorithm, there are at most $n(1-\frac{1}{k})$ points left. Again, since the optimal solution uses k sets, there must some set that covers at least a $\frac{1}{k}$ fraction of the remainder (if we got lucky we might have chosen one of the sets used by the optimal solution and so there are actually $k-1$ sets covering the remainder, but we can't count on that necessarily happening). So, again, since we choose the set that covers the most points remaining, after the second iteration, there are at most $n(1-\frac{1}{k})^2$ points left. More generally, after t rounds, there are at most $n(1-\frac{1}{k})^t$ points left. After t = k ln n rounds, there are at most $n(1-\frac{1}{k})^{k \ln n} < n(\frac{1}{e})^{\ln n} = 1$ points left, which means we must be done.

MAXIMUM FLOW

Overview

- In optimization theory, **maximum flow problems** involve finding a feasible flow through a single-source, single-sink flow network that is maximum.

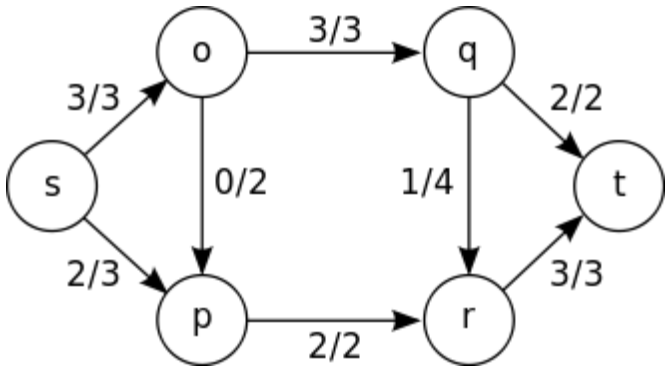


Image courtesy : Wikipedia

Formal Problem Statement

- Let *G=(V,E)* be a network with $s,t \in V$ being the source and the sink of G respectively.
- **Definition**:The **capacity** of an edge is a mapping $c: E \longrightarrow R^+$ denoted by c(u,v) It represents the maximum amount of flow that can pass through an edge.
- **Definition**: A **flow** is a mapping $f: E \longrightarrow R^+$, denoted *by f(u,v),* subject to the following two constraints:

$$\forall (u,v) \in E: \quad f_{uv} \leq c_{uv}$$ (Capacity Constraint)

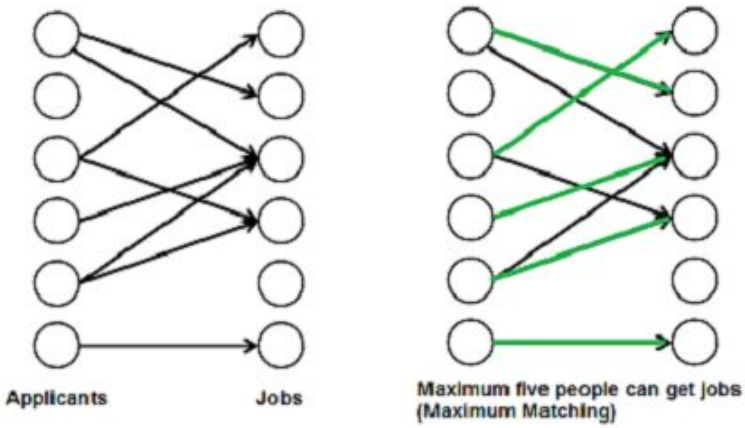
$$\forall v \in V \setminus \{s,t\}: \quad \sum_{\{u:(u,v)\in E\}} f_{uv} = \sum_{\{u:(v,u)\in E\}} f_{vu}.$$ (Conservation Constraint)

Definition. The **value of flow** is defined by $|f| = \sum_{v \in V} f_{sv}$,

PROBLEM : max | *f* |

Applications of the Maximum Flow Problem

Bipartite Graphs Matching Problem

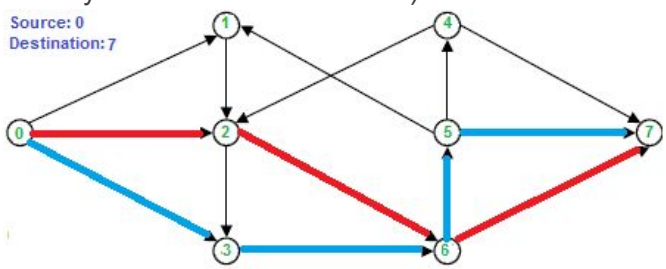


A matching in a Bipartite Graph is a set of the edges chosen in such a way that no two edges share an endpoint. A maximum matching is a matching of maximum size (maximum number of edges). In a maximum matching, if any edge is added to it, it is no longer a matching. There can be more than one maximum matchings for a given Bipartite Graph.

- There are many real world problems that can be formed as Bipartite Matching. For example, consider the following problem:
- *There are M job applicants and N jobs. Each applicant has a subset of jobs that he/she is interested in. Each job opening can only accept one applicant and a job applicant can be appointed for only one job. Find an assignment of jobs to applicants in such that as many applicants as possible get jobs.*

Edge-Disjoint Paths

(Two paths are edge-disjoint if they have no arc in common)

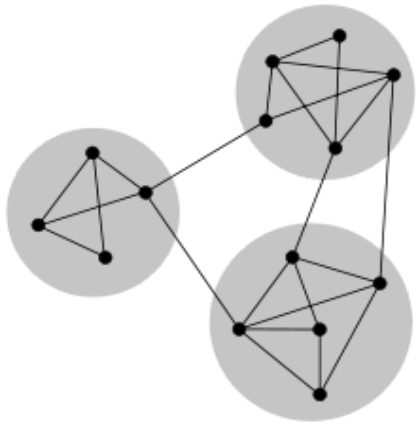


- **INSTANCE**: Directed graph *G(V, E)* with two distinguished nodes *s* and *t.*
- **SOLUTION**: The maximum number of edge-disjoint paths between *s* and *t.*

MAXIMUM FLOW

Network Connectivity

In mathematics and computer science, **connectivity** is one of the basic concepts of graph theory: it asks for the minimum number of elements (nodes or edges) that need to be removed to disconnect the remaining nodes from each other. The connectivity of a graph is an important measure of its resilience as a network.



- **INSTANCE**: Directed graph *G(V, E)* with two distinguished nodes *s* and *t.*
- **SOLUTION**: The minimum number of nodes which needs to be removed *t*o disconnect the graph.

Baseball elimination problem.

- Set of teams *X*.
- Distinguished team *x* *X.*
- Team *i* has won w_i games already.
- Teams *i* and *j* play each other r_{ij} additional times.
- Is there any outcome of the remaining games in which team *x* finishes with the most (or tied for the most) wins?

Image Segmentation Problem

- **INSTANCE:** Pixel graphs *G=(V, E)*, likelihood functions $a, b : V \to R+$, penalty function $p : E \to R+$
- **SOLUTION:** *Optimum labelling* : partition of the pixels into two sets A and B that maximises

$$q(A, B) = \Sigma_{i \in A} a_i + \Sigma_{j \in B} b_i - \Sigma_{i \in A, j \in B} p_{ij}$$

SOLUTIONS TO MAX-FLOW ALGORITHM

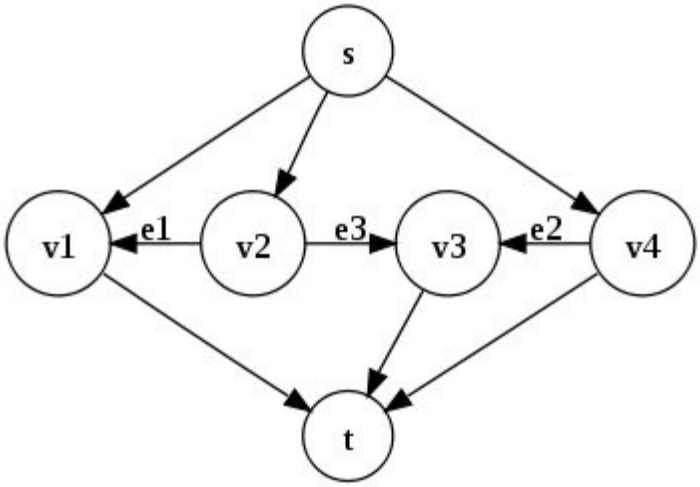
Ford-Fulkerson Algorithm

```
Ford-Fulkerson-algo(G,s,t)
{  f ← 0;
   While (there is s-t path in G_f ) do
// G_f is the residual graph
   {   Let P be an s-t path in G_f
       Let c′  be bottleneck capacity of P;
       For each (x,y) ∈ P do
     {    If (x,y) is a forward edge then
                f(x,y) ← f(x,y) + c′ ;
          Else
                f(y,x) ← f(y,x) - c′ ;
     }
    }  return f;
}
```

The algorithm is only guaranteed to terminate if **<u>if all weights are rational</u>**. Otherwise it is possible that the algorithm will not converge to the maximum value. However, if the algorithm
terminates, it is guaranteed to find the maximum value.

Time Complexity of the Algorithm: $O(E\,|f|_{max})$ where $|f|_{max}$ is the maximum flow in the network.

MAXIMUM FLOW



$w(e_1) = 1$

$w(e_2) = r = (\sqrt{5} - 1)/2$

$w(e_3) = 1$

The algorithm chooses path at random. There are ways to choose path s.t the algorithm does not terminate.

The algorithm is *poor in complexity* and *does not guarantee a solution for real weight edges* but still this algorithm is of much significance. The genius idea of the algorithm was used to extend and modify it to improve the time complexity and solvable for real weights.

The algorithm also provides an elegant proof and insight to one of the important theorems of Graph Theory- *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.

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.

Linear Programming Formulation

PRIMAL(max-flow)

maximise $|f| = \nabla_s$

Subject to

$$\begin{aligned} f_{ij} &\leq c_{ij} && (i,j) \in E \\ \textstyle\sum_{j:(j,i)\in E} f_{ji} - \sum_{j:(i,j)\in E} f_{ij} &\leq 0 && i \in V, i \neq s,t \\ \nabla_s + \textstyle\sum_{j:(j,s)\in E} f_{js} - \sum_{j:(s,j)\in E} f_{sj} &\leq 0 \\ -\nabla_s + \textstyle\sum_{j:(j,t)\in E} f_{jt} - \sum_{j:(t,j)\in E} f_{tj} &\leq 0 \\ f_{ij} &\geq 0 && (i,j) \in E \end{aligned}$$

DUAL(min-cut)

Minimise $\displaystyle\sum_{(i,j)\in E} c_{ij} d_{ij}$

subject to

$$\begin{aligned} d_{ij} - p_i + p_j &\geq 0 && (i,j) \in E \\ p_s - p_t &\geq 1 \\ p_i &\geq 0 && i \in V \\ d_{ij} &\geq 0 && (i,j) \in E \end{aligned}$$

Reduction to Polynomial Time

- 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.

Modified Ford-Fulkerson Algorithm (for integer weighted graphs)

Pseudo code:

```
Ford-Fulkerson-algo(G,s,t)
  f ← 0;
  k ← max-capacity(E);
  While (k ≥ 1) do
   {   While (∃ any s-t path in Gf with capacity ≥ k) do
     {     Pick any such path P;
          For each (x,y) ∈ P do
          {    If (x,y) is a forward edge then
                    f(x,y) ← f(x,y) + c' ;
               Else
                       f(y,x) ← f(y,x) - c' ;
          }
      }
      k ←  k/2 ;
  }
  return f;
```

Analysis of the algorithm:

Consider any edge (x,y), $x \in A$, $y \in \bar{A}$

If $f(x,y) \le c(x,y) - k$

(x,y) must appear as forward edge with $c_f \ge k$ in

y is reachable from s using edges of capacity $\ge k$ in G_f. **A contradiction.**

Consider any edge (x,y), $x \in A$, $y \in \bar{A}$

If $f(x,y) \ge k$

(y,x) appears as a backward edge with $c_f \ge k$ in $\mathbf{G_f}$.

x is reachable from s using edges of residual capacity $\ge k$ in $\mathbf{G_f}$. **A contradiction.**

Consider the flow **f** when there is no **s**-**t** path in $\mathbf{G_f}$ with capacity $\ge k$.

A: set of vertices reachable from **s** with edges of capacity $\ge k$ in $\mathbf{G_f}$

Every edge (x,y) from A to A^{-} carries flow $> c(x,y) - k$

Every edge from A to A^{-} carries flow less than $< k$

$value(f) = f_{out}(A) - f_{in}(A) > |f|_{max} - mk$

Observation 1- Each iteration increases flow by at least k.
We know that there can be only $log\ c_{max}$ iterations for a given k.

Result : Given a flow network $G = (V, E)$, where edge capacities are integers, Algorithm 1 runs in $(O(|E|^2\ log\ c_{max})$ time to compute max s-t flow, where c_{max} is the maximum capacity of any edge.

Edmonds-Karp Algorithm

Edmonds-Karp(**G**,**s**,**t**)

```
{  f ← 0;
   While (there is s-t path in Gf ) do
// Gf is the residual graph
   {   Let P be the shortest s-t path in Gf
       Let c'  be bottleneck capacity of P;
       For each (x,y) ∈ P do
     {    If (x,y) is a forward edge then
                f(x,y) ← f(x,y) + c' ;
           Else
                 f(y,x) ← f(y,x) - c' ;
     }
    }  return f;
}
```

- This algorithm can solve the max-flow problem for **real positive weights** in polynomial time. This is an achievement compared to the previous algorithm which was possible for integer weights only.

  Time Complexity : $O(|E|^2\ |V|)$

Dinic's Algorithm

- This algorithm was developed by Israeli computer scientist Yefim A. Dinitz. Unlike the Edward-Karp Algorithm, this algorithm was not derived from the idea of Ford-Fulkerson Algorithm.
- 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. Dinitz introduced the concept of **blocking flow** in a layered graph which is the key idea used in his algorithm for max-flow.

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

$$G_L = (V, E_L, c_f|_{E_L}, s, t)$$

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

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.

Pseudocode:

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.

- In each phase the algorithms builds a layered graph with breadth-first search on the residual graph. The maximum flow in a layered graph can be calculated in $O(|V||E|)$ time, and the maximum number of the phases is $|V|-1$.

  Time Complexity = $O(|V|^2|E|)$

This algorithm is an improvement over the Edward-Karp Algorithm which takes $O(|E|^2 |V|)$ time to compute max-fow.

Another key advantage: Many applications like Bipartite Matching , computing Edge-Disjoint Paths are special cases of max-flow i.e. with unit edge capacities.

In this case the time complexity of Dinitz's Algorithm reduces to $\boldsymbol{O(|E|.min\{|V|^{2/3}|E|^{1/2}\})}$ complexity which is a huge improvement.

Dinic's Algorithm, where dynamic trees data structure is implemented to speed up the max flow in a layered graph to $O(E \log(V))$ solves the max flow in $O(VE \log(V))$.

Link to IIT KANPUR

- In 1978, a paper was published titled *An $(O|V|^3)$ Algorithm for finding Maximum Flow in Networks.*
  The algorithm described in the paper works for all acyclic graphs with real weight edges and was a considerable improvement over the then known best solution of $O(|E||V|^2)$ time complexity.

  The paper was published by V.M. MALHOTRA, M. PRAMOTH KUMAR and S.N. MAHESHWARI who are former faculty of IIT Kanpur. The algorithm is popularly known as **MKM Algorithm.**

Journey to Milestone

- Another algorithm known as **Push-relabel Algorithm** was designed by Goldberg and Trajan in 1986 which computes the max-flow in $O(V^2E)$ and was reduced to $O(VE\log(V^2/E))$ using dynamic trees.

- After this Trajan along with King and Rao developed a technique to compute the max flow in $O(EV\log_{\frac{E}{V\log V}} V)$ time.
- As recent as a 2013 development by J.Orlin, we now have a combination of algorithms to solve to solve the problem **in a general graph in *O(VE)* time.**

(Orlin + KRT)

- Orlin's algorithm solves the problem in $O(VE)$ time when $E \leq O(V^{\frac{16}{15}-\epsilon})$ and the KRT solves it in $O(VE)$ when $E \geq V^{1+\epsilon}$

References:

[1] Corman H. Thomas, Leiserson E. Charles, Rivest L. Ronald, Stein Clifford "*Introduction to Algorithms*" Second Edition McGrawHill Book Company

[2] Cesari Giovanni *"Divide and Conquer Strategies for Parallel TSP Heuristics"*, Computers Operations Research , Vol.23, No.7, pp 681-694, 1996

[3] del Castillo M. Jose *"A heuristic for the traveling salesman problem based on a continuous approximation"*, Transportation Research Part B33 (1999) 123-152

[4] Gutin Gregory,eo Anders, Zverovich Alexey *"Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the TSP" ,* Discrete Applied Mathematics 117 (2002), 81-86

[5] van der Poort S. Edo, Libura Marek, Sierksma Gerard, vander Veen A. A. Jack *"Solving the k-best traveling salesman problem"*, Computers & Operations Research 26 (1999) 409-425

[6] Valenzuela I. Christine, Jones J. Antonia *"Estimating the Held-Karp lower bound for the geometric TSP* , European Journal of Operational Research 102(1997) 157-175

[7] Zhang Weixiong "*A note on the complexity of the Assymetric Traveling Salesman Problem",* Operation Research Letters 20 (1997) 31-38