**Approximation Algorithms**

The concept of ‘approximation algorithms’ is strongly tied up with what is called an ‘Optimization problem’ which refers to the process of finding the best solution from all achieved solutions. A large number of optimization problems which are required to be solved in practice are NP-hard. In fact, in computer science, there are 2 basic classes of problems: P and NP.

P: includes all the problems that can be solved efficiently, its execution time is bounded by a polynomial which has a finite degree.

NP: nondeterministic polynomial – this represents the class of problems for which one can efficiently verify if a given solution is correct. However, the problem which stands up is whether that solution was found in an efficient way or not.

The concept of NP - hard is somehow making a connection between P and NP because it includes the class of problems for which if one shows that the algorithms used can be efficient, then every problem in NP would be efficiently solved. Therefore, it implies that P=NP.

Coming back to approximation algorithms, I can give a brief definition of them and that would be the following: an algorithm A is said to be an approximation algorithm for an optimization problem P if for any given instance I, it will return an approximate solution.

Basically, an approximation algorithm is guaranteed to run in polynomial time and it is also close to find the optimal solution. However, here comes the problem of P=NP because **you have to prove that the solution given by an approximate algorithm is close to the optimal solution without even knowing what the optimum value is.**

**Comparison with heuristics**: heuristics are based on intuitive steps that may or may not lead to an optimal solution on every given instance. On the other hand, approximation algorithms are thought to be reasonably close to the optimal value, with an approximation up to a small constant factor (like 5%).

**Notations used:**

P – optimization problem

A – an approximation algorithm

I – instance of P

A\*(I) – optimal value for instance P

A(I) – value for the instance P generated by A

= approximation ratio

Minimization: <= (the actual solution obtained is larger than the optimal solution)

Maximization: <= (A\*(I) is larger than the solution obtained)

>=1, =1 => PERFECT approximation.

**Problems in approximation algorithms:**

**Vertex Cover problem: What is the minimum size of the vertex cover?**

**Vertex cover:** a subset of vertices which cover every edge (an edge is covered if one of its endpoints is chosen).

I: an undirected graph G = (V, E).

Possible solution: a subset of the initial set of vertices which would cover all the edges, set C.

**Examples of approaches for vertex cover:**

1. **Approx. optimal solution** - (it doesn’t give the best solution but a nearby solution)

**Step 1:** create an initial empty set cover C and a set E of all the edges of the graph

**Step 2:** while E is not empty choose and arbitrary edge from E, {u, v}

**Step 3:** and add u, v to C

**Step 4:** remove from E all the edges incident on either u or v

**Step 5:** go to step 1until E is empty.

**Performance analysis:**

Optimum vertex cover C\* must cover every edge in a maximal matching M => C\* contains at least one of the endpoints of each edge in M => |C\*| >= |M|

But |C| = 2\* |M | <= 2 \* |C\*|, C\* = optimal solution

Therefore, we have a 2-factor approximation algorithm .

1. **Greedy approach** – choose the ‘nearby optima and hope that this decision will result in a ‘global best’ – worldwide optima.

**Step 1:** find vertex v of maximum degree, add it to C

**Step 2:** delete all edges incident on v (vertex chosen at step 1)

**Step 3:** update the degree of vertices and start again from step 1 until all edges are covered.

Time complexity: O(E+V)

**TSP Approximation** – minimize the cost of the tour

Instance: a complete weighted undirected graph G = (V, E), non-negative edge weights

Possible solution: a cycle that visits each vertex once

Value: is represented by the sum of the weight of the edges in the cycle, we need to keep it minimum.

There are 3 algorithms in computing the solution route:

* Greedy Heuristics
* Nearest Neighbour approach
* Genetic Algorithm

The above three algorithms converge on a particular instance. However, on an instance of almost 1000 randomly distributed cities, the Greedy Heuristics would give the best solution (just very close to the optimal one).

Given any constant c>1, there is no polynomial time algorithm which can approximate TSP within a factor of c. However, if edge weights would satisfy the triangle inequality (in this way it would be more related to the physical word), we could be able to approximate. In this case, TSP becomes Metric TSP which is still NP-Hard but easier to approximate.

w – weight function (the cost) on the edges which satisfies w(u, v) ≤ w(u, x) + w(x, v).

Algorithm steps: (Double tree algorithm)

**Step 1**: select a root vertex in G. Let it be r.

**Step 2:** compute a MST rooted at r, using Prim’s algorithm.

**Step 3:** visit the resulted tree in preorder (root, left, right) and put the vertices in a set

**Step 4**: find a Eulerian tour by applying triangle inequality.

**Step 5:** return the Hamiltonian cycle by shortcutting the tour at step 4.

The MST found is a lower bound of the optimal solution because by deleting an edge from the optimal TSP solution (which is a cycle because the traveling salesman must finish where he was at first) we would obtain a MST.

But by traversing the MST obtained we repeat vertices => it is not a cycle. We also walk each edge twice when we reconstruct the tour. Therefore, it cannot be more than twice the cost of the traveling salesman’s cycle.

The fact that the obtained solution is not a cycle, is corrected by the use of triangle inequality. Instead of going back to the root of a rooted subtree when traversing from the left-most node to the right-most node, we can choose to go on the edge incident on both of them (it exists because G is complete) as the three nodes (leftmost, rightmost and their parent – the root of the subtree) form a triangle and from the above inequality we know that chosen edge between them is of less weight.

In this way, the algorithm is correct because it produces a Hamiltonian circuit.

**Performance analysis:**

Time complexity for Prim’s algorithm: O(ElogV)

T – MST obtained from Prim’s algorithm

C(T) – cost of the tree (C(T) = A(I) but word ‘cost’ is more appropriate here)

C(T) <= C(H\*), H\* = the Hamiltonian cycle which designates the optimal cost value

W – the walk formed by traversing the MST, in this walk we visit each edge twice

* C(W) = 2 \* C(T) <= 2 \* C(H\*)

After applying triangle inequality we obtained a Hamiltonian cycle H

* C(H) < C(W) <= 2 \* C(H\*)
* The cost of the found cycle cannot be greater than the cost of the optimal Hamiltonian cycle.
* **2-factor approximation algorithm**

A different algorithm for TSP is called Nearest addition algorithm (form a trivial cycle by starting with 2 closest nodes and keep adding the vertex that is closest to the cycle. However, this method is also a 2-factor approximation algorithm.