Approximations to NP Solutions

If we are faced with an NP or NP-complete problem the best we can do on a deterministic computer is a solution in exponential time.

Since exponential time solutions are impractical for obvious reasons we could consider approximations to the actual solutions.

Here we look at approximation algorithms for the classic ‘Traveling Salesman Problem’ (TSP).

Traveling Salesman

The idea is, given a set of cities (nodes) that are connected via roads (weighted edges), find the cheapest route through all the cities (find a Hamiltonian path that minimizes the sum of the weights in the path). Formally,

$TSP = \{\langle G, s, t, w\rangle | G$ is directed weighted graph with a minimal Hamiltonian path of weight w from s to $t\}$.

Traveling Salesman

Theorem:

$$TSP \in NP\text{-hard}.$$

Proof: Note, a problem is NP-hard if every $L \in NP$ can be reduced to it in polynomial time but the problem itself is not in NP. No known NP solution exists for TSP (NP problems have polynomial time verifiers; in TSP it is not possible to verify a certificate in polynomial time). It remains to show that all $L \in NP$ reduce to it in polynomial time. We will show this by a polynomial time reduction f from $HAMPATH$ to TSP,

$$\langle G, s, t\rangle \in HAMPATH \text{ iff } f(\langle G, s, t\rangle) \in TSP,$$

where $f(\langle G, s, t\rangle) = \langle G', s, t, m\rangle$ with G' the graph G with a weight of 1 on all of its edges and m the number of nodes in G. Clearly, the reduction runs in polynomial time. We verify the reduction condition by first observing that a Hamiltonian path gives rise to a minimal traveling salesman circuit by the virtue that all Hamiltonian paths in G' have the same cost. The converse also holds, if we have a traveling salesman circuit this implies that we have a Hamiltonian path. □

Approximation Algorithms

Approximation algorithms are an approach to attacking difficult optimization problems. Approximation algorithms are often associated with NP-hard problems. Since it is unlikely that there can ever be efficient (Polynomial Time) exact algorithms solving NP-complete/hard problems, one settles for non-optimal solutions, but requires them to be found in polynomial time.

Unlike heuristics, which usually only find reasonably good solutions reasonably fast, one wants provable solution quality and provable run time bounds. Ideally, the approximation is optimal up to a small constant factor (say within 5% of the optimal solution). It should be noted that approximation algorithms are increasingly being used for problems where polynomial algorithms are known but are too expensive due to the sizes of the data sets.

A Heuristic

A good heuristic is the ***nearest neighbor*** algorithm.[a]

```
$M$ = "On input $\langle G \rangle$, where $G$ is a directed weighted graph:
1. Select node $s$ as the starting point. Set $s$ as current node.
2. While there are unvisited nodes reachable from current node repeat:
3.     Find out the lightest edge connecting current node and a unvisited node $v$.
4.     Set current node to be $v$.
5.     Mark $v$ as visited.
6. If all nodes have been visited, accept; otherwise, reject."
```

In general, this heuristic provides a circuit with length $log(m) * t$, where m is the number of nodes in the graph and t the optimal circuit. But this bound is not guaranteed.

[a] `http://en.wikipedia.org/wiki/Travelling_salesman_problem`

An Approximation Algorithm

Definition: (Triangle Inequality) The property that a complete weighted graph satisfies $weight(u,v) \leq weight(u,w) + weight(w,v)$ for all vertices u, v, w. (Informally, the graph has no short cuts.)

(source: www.itl.nist.gov)

An Approximation Algorithm

Minimum Spanning Tree: The algorithm continuously increases the size of a tree starting with a single vertex until it spans all the vertices.

- Input: A connected weighted graph with vertices V and edges E.
- Initialize: $V_{\text{new}} = \{s\}$, where s is an arbitrary node (starting point) from V; $E_{\text{new}} = \{\}$.
- Repeat until $V_{\text{new}} = V$:
  - Choose edge (u, v) from E with minimal weight such that u is in V_{new} and v is not (if there are multiple edges with the same weight, choose arbitrarily)
  - Add v to V_{new}, add (u, v) to E_{new}
- Output: V_{new} and E_{new} describe a minimal spanning tree.

Runs in $O(V^2)$ time.

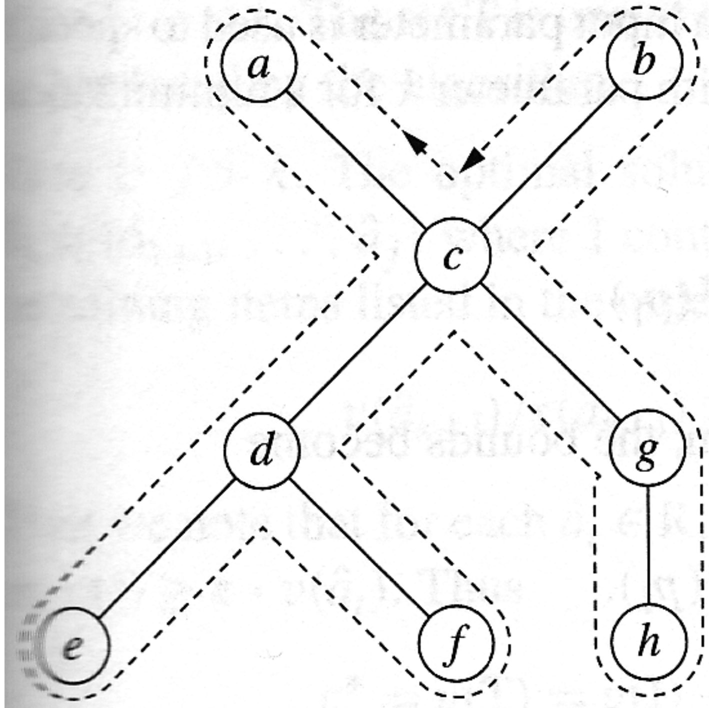
An Approximation Algorithm

Assume that the directed graph G is fully connected and the triangle inequality is satisfied. Then we can use the following algorithm to compute an approximate solution to the TSP problem.

```
$M$ = "On input $\langle G \rangle$, where $G$ is a directed fully connected weighted graph:
1. Select node $s$ as the start node for the circuit.
2. Construct a minimum-cost spanning tree rooted in $s$.
3. Construct a sequence of nodes visited by a preorder traversal of the
   spanning tree.
4. Delete nodes that occur more than once in the sequence.
5. *accept*."
```

The triangle equality and the fully-connectedness assures that we can build the circuit. It can be shown that the cost of the computed circuit is bounded by twice the cost of the minimal spanning tree.

An Approximation Algorithm



Node visit sequence: *c, a, c, d, e, d, f, d, c, g, h, g,*

Tour: *c, a, d, e, f, g, h, b, c*

Approximations to P Solutions: Probabilistic Machines

Here is our first extension to traditional TM's

Definition: A **probabilistic Turing machine** M is a type of nondeterministic Turing machine in which each nondeterministic step is called a **coin-flip step** and has two legal next moves. We assign a probability to each branch b of M's computation on input w as follows. Define the probability of branch b to be

$$Pr[b] = 2^{-i},$$

where i is the number of coin-flip steps that occur in branch b. Define the probability that M accepts w to be

$$Pr[M \text{ accepts } w] = \sum_{b \text{ is an accepting branch}} Pr[b].$$

Furthermore,

$$Pr[M \text{ rejects } w] = 1 - Pr[M \text{ accepts } w].$$

Probabilistic Machines

Our definition implies the following: When a probabilistic TM recognizes a language then it must accept all strings in the language and reject all strings not in the language with the exception that now we allow the machine to make errors with some small probability.

For $0 \leq \epsilon < 1/2$ we say that M **recognizes language** A **with error probability** ϵ if

1. $w \in A$ implies $Pr[M \text{ accepts } w] \geq 1 - \epsilon$, and
2. $w \notin A$ implies $Pr[M \text{ rejects } w] \geq 1 - \epsilon$.

That is, the probability that we obtain a wrong answer is at most ϵ.

Amplification Lemma

The following lemma allows us to make the error on a probabilistic machine as small as desired.

Lemma: (Amplification Lemma) Let ϵ be a fixed constant strictly between 0 and $1/2$, then for any polynomial $f(n)$ a probabilistic polynomial time TM M_1 that operates with error probability ϵ has an equivalent probabilistic polynomial time TM M_2 that operates with an error probability of $2^{-f(n)}$.

Proof Sketch: The machine M_2 runs the machine M_1 k number of times where k is picked such that the error probability $2^{-f(n)}$ holds. The outcome of the machine is determined by majority vote.

M_2 = "On input w:

1. Calculate k (see below).
2. Run $2k$ independent simulations of M_1 on w.
3. If most runs accept, the *accept*; otherwise *reject*.

We can compute k as

$$k \geq \frac{f(n)}{\log_2(4\epsilon(1-\epsilon))},$$

where $f(n) \geq 1$.

Monte-Carlo Algorithms

Given a language A recognized by a probabilistic TM M, we say that M is a Monte-Carlo algorithm, if

1. $w \in A$ implies $Pr[M$ accepts $w] \geq 1/2$, and
2. $w \notin A$ implies $Pr[M$ rejects $w] = 1$.

This gives rise to a new complexity class called **Randomized Polynomial** time algorithms, RP.

Definition: RP is the class of languages that are recognized by probabilistic polynomial time TMs where the inputs in the language are accepted with a probability of at least $1/2$ and input not in the language are rejected with probability of 1.

From a theoretical point of view these algorithms are barely distinguishable from P. However, from a practical point of view Monte-Carlo algorithms allow us to compute solutions in P that are otherwise too complex.

Monte-Carlo Algorithms

Example: The Triangle decision problem. A graph is in Triangle if it contains a 3-clique. This problem is in P with $O(n^3)$. We can construct a Monte-Carlo algorithm M for this problem: (a) pick an edge $(v, u) \in E$ with $v, u \in V$ and a vertex $p \in V$ such that $p \neq v$ and $p \neq u$. If (u, v, p) form a triangle accept otherwise reject. (b) perform step (a) k times and accept if at least one of the iterations accepts.

In order to verify that this algorithm is Monte-Carlo we have to verify the conditions

1. $\langle G \rangle \in$ Triangle implies $Pr[M$ accepts $\langle G \rangle] \geq 1/2$, and
2. $\langle G \rangle \notin$ Triangle implies $Pr[M$ rejects $\langle G \rangle] = 1$.

Condition 1 can be shown to be fulfilled by the algorithm if we pick $k = |E|(|V| - 2)/3$.

Condition 2 is easily verified; the algorithm cannot accept a graph with no triangle, all experiments fail.