CS 4980: Capstone Research Notes

Professor Mark Floryan,
compiled by Grady Hollar

Fall 2025

1 Basic Definitions

What is an approximation algorithm?

2 Basic Examples

Vertex cover, set cover?

3 The Traveling Salesperson Problem

3.1 Metric TSP

The Traveling Salesperson Problem (TSP) is perhaps one of the best known NP-hard optimization problems in all of computer science. Informally, it asks, given a list of cities and distances between each pair of cities, what is the shortest route that visits every city exactly once and starts and ends at the same city? In the version of the problem we consider, we will further require that the *triangle inequality* holds for all distances between cities. This is an important distinction that we will touch on more later. Formally, this problem is known as *Metric*-TSP, and is proposed as follows:

Optimization Problem (Metric-TSP) Given a complete undirected graph $G = (V, E)$ and cost function $c : E \to \mathbb{Q}^+$ such that the triangle inequality holds for all $u, v, w \in V$:

$$c(u, v) \leq c(u, w) + c(w, v),$$

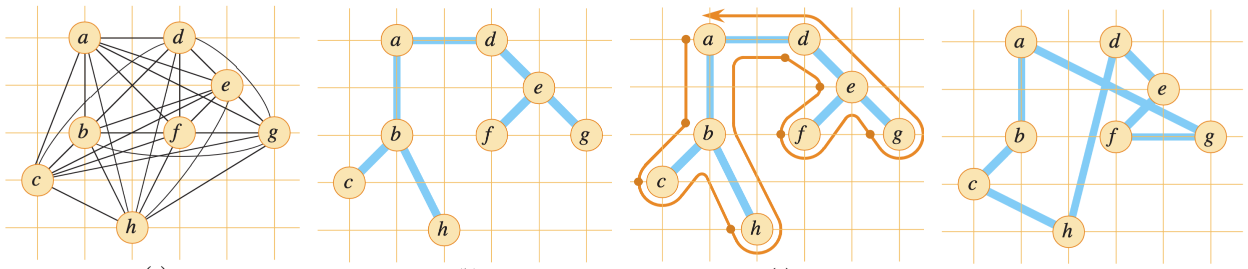
find a minimum cost cycle in G that visits every vertex exactly once.

In this section, we'll devise a 2-factor approximation algorithm for Metric-TSP. Our approach revolves around finding a minimum spanning tree of G and constructing a cycle using the MST. The algorithm works as follows:

Algorithm Approx-TSP(G, c)

```
  Compute a minimum spanning tree $T$ of $G$.
  Perform a preorder traversal $\mathcal{P}$ of $T$.
  Construct a Hamiltonian cycle $H$ by listing the vertices of $G$ in the order of their first appearance in $\mathcal{P}$.
  return $H$
```

Below is a depiction of the algorithm in action. Figure (a) is the original complete graph G, with Figure (b) showing a minimum spanning tree T of G. Figure (c) outlines the preorder traversal $\mathcal{P}$ of T. Lastly, Figure (d) is the resulting Hamiltonian cycle H constructed from $\mathcal{P}$, where the backtracking steps have been "short-cut" for more direct paths.



(a) (b) (c) (d)

To place an approximation ratio on the cost of the resulting cycle H, we begin by first establishing a bound on the cost of the traversal $\mathcal{P}$. Then, since the triangle inequality holds in G, this "short-cutting" step we use to construct H will not increase the cost of the traversal, and the same bound will hold for H. Let's flesh out this approach and prove our claimed approximation ratio.

Theorem 3.1. Approx-TSP is a 2-factor approximation for the Metric-TSP problem.

Proof. There are numerous well-established and efficient algorithms to compute minimum spanning trees, so the algorithm clearly runs in polynomial time. Now, lets first consider the optimal tour O. Since O visits each vertex of G exactly once, deleting any edge from O will produce a spanning tree of G. Since T is a *minimum* spanning tree, it follows that $c(T) \leq c(O)$.

Next, note that since the traversal $\mathcal{P}$ contains each edge of T exactly twice, the cost of $\mathcal{P}$ is precisely double that of the cost of T, or rather, $c(\mathcal{P}) = 2 \cdot c(T)$. Finally, as noted above, the triangle inequality guarantees that $c(H) \leq c(\mathcal{P})$. Combining these inequalities, we have

$$c(H) \leq c(\mathcal{P}) = 2 \cdot c(T) \leq 2 \cdot c(O),$$

which establishes the desired approximation ratio. □

3.1.1 Improving the ratio to 3/2

Another way to describe the above algorithm is through *Eulerian tours*. Recall that an Eulerian tour of a graph is a cycle that visits every edge exactly once, allowing for revisited vertices. An equivalent formulation of Approx-TSP finds an MST, doubles each edge in the tree, performs an Eulerian tour on the doubled tree, then short-cuts the tour to find a Hamiltonian cycle. Is there a cheaper subgraph to perform an Eulerian tour on than the one obtained by doubling every edge in the MST?

A key result in graph theory is that a connected graph has an Eulerian tour if and only if every vertex has even degree. So, we only need to concern ourselves with somehow "remedying" the odd degree vertices in the MST to obtain a subgraph with an Eulerian tour. Using this idea, it's possible to produce an algorithm with a 3/2 approximation factor for Metric-TSP. Readers interested in the details are encouraged to read Section 3.2 in Vazirani's *Approximation Algorithms*.

Whether the 3/2 factor can be improved further is currently one of the biggest open problems within the field of approximation algorithms.

3.2 Inapproximability of General TSP

We devised our approximations for TSP under the assumption that the triangle inequality holds on the graph's edge weights. While this holds true practically for many graphs, e.g. a network modeling roads and intersections where edge weights are lengths of roads, in complete generality, this is quite a big assumption. Can we still approximate TSP without this assumption?

As it turns out, the answer is almost certainly not. More formally, we'll prove that general TSP cannot be approximated at all, assuming $P \neq NP$. Results such as these are known as *inapproximability* or *hardness of approximation* results. These results show that, for some NP-hard optimization problems, approximating them past a certain factor would prove that $P = NP$. This is quite interesting, since as we will see later, other NP-hard optimization problems can be approximated arbitrarily well. In some sense, some problems are much harder to approximate than others.

The general strategy for proving hardness of approximation results is to assume the existence of an approximation algorithm of a certain degree and to use the algorithm to create a decider for some known NP-hard problem. In the case of TSP, we will show that an approximation algorithm of *any* factor can be used to decide the Hamiltonian cycle problem in polynomial time.

Theorem 3.2. *For any function $\rho(n) \geq 1$, TSP cannot be approximated within a factor of $\rho(n)$, unless $P = NP$.*

Proof. Assume for contradiction there is a factor $\rho(n)$ polynomial time approximation algorithm, A, for the general TSP problem. The key idea is a reduction from the Hamiltonian cycle problem to the TSP problem that takes a graph $G = (V, E)$ on n vertices and transforms it into a complete graph G' on n vertices such that:

(1) If G *does* have a Hamiltonian cycle, the cost of an optimal TSP tour in G' is n, and

(2) If G *doesn't* have a Hamiltonian cycle, the cost of an optimal TSP tour in G' is strictly greater than $\rho(n) \cdot n$.

What happens when we run A on our new graph G'? Since A must produce a solution within $\rho(n)$ of the optimal, it must return a tour of cost $\leq \rho(n) \cdot n$ in the first case and $> \rho(n) \cdot n$ in the second. To decide whether the original graph G had a Hamiltonian cycle, we simply need to compute the cost of the given TSP tour and compare it with $\rho(n) \cdot n$. So, if we can find such a reduction, A can be used to solve the Hamiltonian cycle problem in polynomial time.

The reduction is quite simple. We construct G' by completing the graph G on the same vertex set V, with edge weights defined as follows: (notice that these edge weights will almost certainly not satisfy the triangle inequality!)

$$c(u, v) = \begin{cases} 1, & (u, v) \in E \\ \rho(n) \cdot n, & (u, v) \notin E. \end{cases}$$

If G contains a Hamiltonian cycle, then the optimal tour on G' is simply this cycle, which has cost n. If G does not contain a Hamiltonian cycle, the optimal tour on G' must use some edge *not* in G, which has cost $\rho(n) \cdot n$, making the total tour cost strictly greater than $\rho(n) \cdot n$. $\square$

4 Randomization and LP Methods

4.1 A simple MAX-3SAT algorithm

Let's take a moment to remember the most fundamental NP-complete problem of them all: the *satisfiability* (or SAT) *problem.* Recall that a Boolean formula in k-conjunctive normal form (k-CNF) consists of clauses connected by logical and's, where each clause consists of exactly k distinct literals connected by logical or's. For example,

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee \neg x_4 \vee x_5) \wedge (\neg x_3 \vee x_4 \vee x_6) \wedge (x_1 \vee x_3 \vee x_5)$$

is a 3-CNF formula with 4 clauses and 6 variables. Given a k-CNF formula, the SAT problem asks whether we can find an assignment for its variables such that the formula evaluates to true. In the optimization version of this decision problem, we'll instead try to satisfy as many clauses in the formula as possible.

Optimization Problem (MAX-3SAT) Given a 3-CNF formula ϕ, find an assignment of ϕ that satisfies the largest number of clauses.[1]

How good of a solution could we get to the above problem by setting each variable in the given formula completely randomly? We'll soon see that, on average, this simple approach will surprisingly get us close to an optimal solution—quite close, in fact! But what do we mean by "on average?" We need to formalize what it means to have an approximation factor when our algorithm involves an element of randomization.

Definition (Randomized Approximation Algorithm). We say that a randomized algorithm for an optimization problem has an approximation ratio of $\rho(n)$ if, for any input of size n, the *expected cost* of the solution produced by the randomized algorithm is within a factor of $\rho(n)$ of the cost of an optimal solution.

To see how we calculate this expected cost in practice, let's prove that our proposed seemingly naive approach is, interestingly, an 8/7-approximation for the MAX-3SAT problem.

```
Algorithm  Approx-MAX-3SAT($\phi$)
  for $x_i \in \phi$ do
      Assign $x_i$ randomly to be 0 or 1 with equal probability 1/2.
  return the assignment
```

Theorem 2.1. Approx-MAX-3SAT is an 8/7-approximation for MAX-3SAT.

Proof. Let ϕ be a 3-CNF formula with n variables $x_1, x_2, \ldots, x_n$ and m clauses. For $1 \leq i \leq m$, define the random variable

$$Y_i = \begin{cases} 1, & \text{clause } i \text{ is satisfied} \\ 0, & \text{otherwise.} \end{cases}$$

Then, the number of clauses satisfied overall is modeled by the random variable defined by

$$Y = \sum_{i=1}^{m} Y_i,$$

so that the expected cost of the algorithm will be exactly $\mathbb{E}[Y]$. Since each literal is set to 1 with probability 1/2 and 0 with probability 1/2, and a clause is not satisfied only if all three of its literals are set to 0, we have

$$P[Y_i = 0] = (1/2)^3 = 1/8 \quad \Longrightarrow \quad P[Y_i = 1] = 1 - 1/8 = 7/8.$$

Computing $\mathbb{E}[Y_i]$ then gives us

$$\mathbb{E}[Y_i] = 1 \cdot 7/8 + 0 \cdot 1/8 = 7/8.$$

Now we can use the familiar properties of expected values along with the above calculations to see that

$$\begin{aligned} \mathbb{E}[Y] &= \mathbb{E}\left[\sum_{i=1}^{m} Y_i\right] \\ &= \sum_{i=1}^{m} \mathbb{E}[Y_i] \\ &= \sum_{i=1}^{m} 7/8 \\ &= 7m/8. \end{aligned}$$

Since the maximum amount of clauses that can be satisfied in ϕ is m, the approximation ratio is at most $m/(7m/8) = 8/7$. □

[1] In classical literature, the definition of a k-CNF formula only requires each clause to contain *at most* k literals, not exactly k. Adding in this extra condition as we have is formally known as MAX-EkSAT, with the E standing for "exactly". It turns out, however, that the usual formal version of the MAX-3SAT problem *still* admits an approximate solution with a factor identical to the one we are about to show for MAX-E3SAT, but the proof of this is far beyond the scope of this writeup. The interested reader may wish to look into the "Karloff-Zwick algorithm".

The keen reader may have realized that in our proof we have not addressed an important possibility: what if a clause contains both a variable and its negation? Surely this will change the probabilities we calculated, and in turn the approximation factor. Let's show that, in fact, there is no need for such worries.

Proposition 2.2. Without assuming no clause contains a variable and its negation, Algorithm 1 is *still* an 8/7-approximation for MAX-3SAT.

Proof. Suppose that k of ϕ's clauses contain both a variable and its negation, and m clauses do not. For each of these remaining m clauses, we'll define the same random variable Y_i as we did before. Now, notice that for each of the k clauses containing a variable and its negation, the clause will be satisfied *no matter what assignment* for ϕ is chosen. So, the total number of satisfied clauses in ϕ is exactly

$$Y = Y_1 + Y_2 + \cdots + Y_m + k.$$

The calculation of each $\mathbb{E}[Y_i]$ is the same as before, and so

$$\mathbb{E}[Y] = \mathbb{E}\left[\sum_{i=1}^{m} Y_i + k\right] = \sum_{i=1}^{m} \mathbb{E}\left[Y_i\right] + k = 7m/8 + k.$$

Since $m + k$ is the maximum number of clauses that can be satisfied, the approximation ratio is at most

$$\frac{m+k}{7m/8+k} \leq \frac{m+k}{7m/8+7k/8} = \frac{8}{7}.$$

□

Interestingly, it has been shown that, assuming $P \neq NP$, no polynomial-time algorithm for MAX-3SAT can achieve an approximation ratio strictly better than 8/7.[2] So, in the end, our somewhat naive approach gave us essentially the best solution we could hope for!

4.2 Weighted vertex cover and LP rounding

Optimization Problem (Min-Weight Vertex Cover) Given an undirected graph $G = (V, E)$ and a weight function on vertices $w : V \to \mathbb{Q}^+$, find a vertex cover $C \subseteq V$ of minimum weight $w(C) = \sum_{v \in C} w(v)$.

Consider the following linear program:

$$\begin{aligned}
\text{minimize} \quad & \sum_{v \in V} w(v) \cdot x_v \\
\text{subject to} \quad & x_u + x_v \geq 1 \quad \forall (u,v) \in E \\
& x_v \leq 1 \quad \forall v \in V \\
& x_v \geq 0 \quad \forall v \in V
\end{aligned}$$

Algorithm APPROX-MIN-WEIGHT-VC(G, w)

```
$C = \emptyset$
Compute an optimal solution $\overline{x}$ to $G$'s associate linear program.
for $v \in G.V$ do
    if $\overline{x}_v \geq 1/2$ then
        $C = C \cup \{v\}$
return $C$
```

Theorem 2.2. APPROX-MIN-WEIGHT-VC is a 2-approximation algorithm for the minimum weight vertex cover problem.

[2]The full proof of this incredible fact is far beyond the scope of this writeup, but the interested reader can begin to learn more here.

Proof. Want to show:

Why is C a cover?

Compare an optimal cover C^* to the objective function value for optimal solution of the LP z^*. Obtain $z^* \leq w(C^*)$. (C^* is a feasible solution to the LP)

Obtain $z^* \geq w(C)/2$. Key step:

$$z^* \geq \sum_{v \in V : \overline{x}_v \geq 1/2} w(v) \cdot \overline{x}_v$$

Combine inequalities to get $w(C) \leq 2w(C^*)$.

□

5 A Fully Polynomial Time Approximation Scheme

5.1 PTAS's and FPTAS's

5.2 Knapsack

Optimization Problem (Knapsack) Given a collection $A = \{a_1, \ldots, a_n\}$ of n objects, a weight function $w : A \to \mathbb{N}$, a profit function $p : A \to \mathbb{N}$, and a "knapsack capacity" $W \in \mathbb{N}$, find a subset S of A whose total weight is bounded by W and whose total profit $p(S)$ is maximized.

Let's start by proposing a dynamic programming solution to the problem. We'll denote by $DP(i, \ell)$ the weight of the lightest subset, denoted $S_{i,\ell}$, we can obtain from $\{a_1, \ldots, a_i\}$ whose profit is exactly ℓ. Then, the answer to the Knapsack problem is precisely the subset associated with the value $\max\{\ell : DP(n, \ell) \leq W\}$.

Let $P = \max\{p(a) : a \in A, p(a) \leq W\}$ be the value of the most profitable object in A that can fit in the knapsack. Clearly, nP is an upper bound for the profit of any solution we can find, so we only need to fill in values of the DP table for $1 \leq i \leq n$ and $0 \leq \ell \leq nP$. At each stage, we may either add or leave out the a_i'th element from the knapsack. If we leave it out, $DP(i, \ell)$ remains the same as $DP(i-1, \ell)$. If we add it, we add its weight to $DP(i-1, \ell - p(a_i))$ (the previous optimal solution with respect to the *remaining* profit). We can model this choice with the following recurrence:

$$DP(i, \ell) = \begin{cases} \min\{DP(i-1, \ell),\ w(a_i) + DP(i-1, \ell - p(a_i))\} & \text{if } p(a_i) \leq \ell \\ DP(i-1, \ell) & \text{otherwise,} \end{cases}$$

which gives rise to the following bottom-up DP algorithm.

```
Algorithm KNAPSACK-DP($A, w, p, W, P$)
  Initialize $DP(1, \ell)$ for $\ell \in \{0, 1, \ldots, nP\}$.

  for $i = 2$ to $n$ do
      for $\ell = 0$ to $nP$ do
          $add = \infty$
          if $p(a_i) \leq \ell$ then
              $add = w(a_i) + DP(i-1, \ell - p(a_i))$
          $leaveOut = DP(i-1, \ell)$
          $DP(i, \ell) = \min\{add,\ leaveOut\}$
          Store the corresponding $S_{i,\ell}$.

  $M = \max\{\ell : DP(n, \ell) \leq W\}$
  return $S_{n,M}$
```

But wait... Doesn't this solve the Knapsack problem in polynomial time? Not quite. Notice that the runtime of $O(n^2 P)$ depends on the maximum profit P. But since this profit could be arbitrarily large with respect to n (potentially $n! \cdot 2^{n^2}$ large, for instance!), this is not a true polynomial-bounded runtime. When an algorithm is polynomially bounded in both its input size and its *numerical parameters* like the one above, we call it a *psuedo-polynomial time algorithm.*

If it just so happened that $P = q(n)$ for some polynomial q, then our algorithm will in fact run in polynomial time. To obtain an FPTAS, we are going to exploit precisely this fact by ignoring a certain number of least significant bits of the profits of our objects to ensure P will be small enough.

Algorithm KNAPSACK-FPTAS$(A, w, p, W, P, \varepsilon)$

```
  $K = \varepsilon \frac{P}{n}$
  Define a new profit function $p'$ by $p'(a_i) = \lfloor \frac{p(a_i)}{K} \rfloor$ for each $a_i$.
  $S =$ KNAPSACK-DP$(A, w, p', W, P)$
  return $S$
```

To show that this is indeed an FPTAS, we begin by placing a bound on its approximation ratio.

Lemma 5.1. Fix some $0 < \varepsilon < 1$. Let S be the solution returned by KNAPSACK-FPTAS run with parameter ε, and let O be the true optimal solution. Then,

$$\frac{p(O)}{p(S)} \leq \frac{1}{1-\varepsilon}.$$

Proof. First, due to rounding, we have that

$$p(S) = \sum_{a \in S} p(a) \geq \sum_{a \in S} K \cdot \left\lfloor \frac{p(a)}{K} \right\rfloor.$$

Next, note that since S is an optimal solution with respect to the new profits $p'(a) = \lfloor \frac{p(a)}{K} \rfloor$, it will dominate the new profit of the original optimal solution O:

$$\sum_{a \in S} K \cdot \left\lfloor \frac{p(a)}{K} \right\rfloor \geq \sum_{a \in O} K \cdot \left\lfloor \frac{p(a)}{K} \right\rfloor.$$

Lastly, note that for any single object a, due to rounding again, $K \cdot \lfloor \frac{p(a)}{K} \rfloor$ will never be smaller than $p(a)$ by more than K. That is,

$$\sum_{a \in O} K \cdot \left\lfloor \frac{p(a)}{K} \right\rfloor \geq \sum_{a \in O} \big(p(a) - K \big).$$

Putting the three previous inequalities together gives us

$$p(S) \geq \sum_{a \in O} \big(p(a) - K \big) \geq p(O) - nK.$$

Finally, by noting that $P \leq p(O)$, since naively picking the single most profitable object is a feasible solution, we see that

$$p(S) \geq p(O) - nK = p(O) - \varepsilon P \geq p(O) \cdot (1 - \varepsilon),$$

which is exactly the inequality we wanted to show. □

With the lemma done, we are ready for the final proof.

Theorem 5.2. KNAPSACK-FPTAS is a fully polynomial time approximation scheme for the Knapsack problem.

Proof. Given $\varepsilon > 0$, we'll run our algorithm with a new parameter $\delta = \frac{\varepsilon}{1+\varepsilon} < 1$. We can use the bound from Lemma 5.1 to see that

$$\frac{p(O)}{p(S)} \leq \frac{1}{1-\delta} = \frac{1}{1 - \varepsilon/(1+\varepsilon)} = 1 + \varepsilon,$$

so that our solution is within $1 + \varepsilon$ of the optimal.

Next, with the adjusted profits and new parameter δ, the running time of the call to Knapsack-DP will be

$$O\left(n^2\left\lfloor\frac{P}{K}\right\rfloor\right) = O\left(n^2\left\lfloor\frac{n}{\delta}\right\rfloor\right) = O\left(n^3 \cdot \frac{1+\varepsilon}{\varepsilon}\right) = O\left(n^3 \cdot \frac{1}{\varepsilon} + n^3\right),$$

which is polynomial in n and $1/\varepsilon$. So, our algorithm is indeed a fully polynomial time approximation scheme for the Knapsack problem! □