CS3100

Approximation
Announcements

- HW out + oral grading next Friday
Hard Problems

Apparently, the world is full of them:
- some impossible
  - ie
- others just slow

What to do?
- Approximate
- Randomization
Example: Load Balancing

- n jobs, each with a running time $T_{i,1...n}$
- m machines available on which to run them

Goal: Compute an assignment $A[i,1...n]$ where job $i$ gets assigned to some machine $i \in [1...m]$

i.e. $A[j] = i$

Picture:

```
\begin{array}{c}
1 & 2 \\
1 & 1 & 15 \\
1 & 2 & 10 \\
\end{array}
```
Natural Goal:
Finish as early as possible!

Makespan: max time any machine is running jobs:

\[
\text{makespan}(A) = \max_i \left( \sum_{j: A[i,j]=i} T_{i,j} \right)
\]

Goal:
Minimize makespan:

\[
\min_A \max_i \left( \sum_{j: A[i,j]=i} T_{i,j} \right)
\]
This is NP-hard. Why?

Reduce partition to this:

Given list \( S = \{s_1, \ldots, s_n\} \)

\( \rightarrow \) run \( n \) jobs with \( T[j] = s_j \)

\( + \) set \( m = 2 \).

ask for makespan of value \( \frac{3}{5} \)
Approximating

What seems a natural strategy?

Greed!

Possible heuristic:

A heuristic technique (/ˈhjuːrɪstɪk/; Ancient Greek: εὑρίσκω, *find* or *discover*), often called simply a heuristic, is any approach to problem solving, learning, or discovery that employs a practical method not guaranteed to be optimal or perfect, but sufficient for the immediate goals. Where finding an optimal solution is impossible or impractical, heuristic methods can be used to speed up the process of finding a satisfactory solution. Heuristics can be mental shortcuts that ease the cognitive load of making a decision. Examples of this method include using a rule of thumb, an educated guess, an intuitive judgment, guesstimate, stereotyping, profiling, or common sense.

Consider jobs 1 at a time
+ assign to current "emptiest" machine.
Algorithm:

```
GREEDYLOADBALANCE(T[1..n], m):
    for i ← 1 to m
        Total[i] ← 0
    for j ← 1 to n
        mini ← arg min_i Total[i]
        A[j] ← mini
        Total[mini] ← Total[mini] + T[j]
    return A[1..m]
```

Runtime:

\[ m + n(m + 1) = O(n m) \]

If you do Total[i]'s in a heap:

\[ O(n \log m) \]
Claim: The makespan of this greedy algorithm is at most twice the optimal solution.

\[ \text{pf: start w/ 2 observations:} \]

1. \[ \text{OPT} \geq \max_j \{ \text{maximal makespan} \} \]

2. \[ \text{OPT} \geq \text{average job length} \]
   \[ = \frac{1}{n} \sum_{j=1}^{n} T[j] \]
Now consider machine w/ largest makespan in greedy alg L2 machine $i$.

Let $j$ be last job assigned to machine $i$.

Picture:

$M_1, M_2, M_i, ..., M_m$
Better picture:

Proof that GreedyLoadBalance is a 2-approximation algorithm

Objective 1

Goal:

\[
\text{Total } [i] - \text{Total } [j] \leq \text{OPT}
\]

\(j\)'s makespan with \(j\) removed

When \(j\) was assigned, \(i\) had lowest makespan

\[
\text{Total } [i] - \text{Total } [j] \leq \text{Total } [k]
\]

\(\geq\) had to be less than or equal to average

by \(\ominus\), \(\text{OPT} \geq \text{average}\)
Q: Could this be optimal?
   (Answer: NO!
   Possibly on hw...)

Note: This is actually an online algorithm

Why might this be a useful observation?
Can we do better if given input offline? Yes!

\[
\text{SortedGreedyLoadBalance}(T[1..n], m):
\rightarrow \text{sort } T \text{ in decreasing order}
\rightarrow \text{return GreedyLoadBalance}(T, m)
\]

\textbf{Claim:} Makespan of above is \( \leq \frac{3}{2} \cdot \text{OPT} \).

\textbf{Proof:} 2 cases:

\[ n \leq m : \text{ (easy case) \quad one per machine} \]
\[ \Rightarrow \text{greedy = OPT} \quad (\leq \frac{3}{2}\text{OPT}) \]
Otherwise: \( n > m \).

Consider \( i = j \) as before:

\[
\text{Still have: } \text{Total}[i] - T[j] \leq \text{OPT}.
\]

Now: in any schedule, some machine must have \( 2 \)
of the first \( m+1 \) jobs.

\[
(\Rightarrow \text{say } k + \ell \leq m+1)
\]

\[
T[k] + T[\ell] \leq \text{OPT}
\]

So:

\[
T[j] \leq T[m+1] \leq T[\max \{k, \ell\}] \\
(\text{since sorted})
\]

\[
\leq \frac{\text{OPT}}{a}
\]
Defs for Approx:

Let \( \text{OPT}(x) \) = value of optimal solution

\( A(x) \) = value of solution computed by algorithm \( A \)

\( A \) is an \( \alpha(n) \)-approximation algorithm if:

\[
\frac{\text{OPT}(x)}{A(x)} \leq \alpha(n)
\]

and

\[
\frac{A(x)}{\text{OPT}(x)} \leq \alpha(n)
\]

\( \alpha(n) \) is called the approximation factor.
So greedy load balancing:

\[ A(x) \leq 2 \text{OPT}(x) \]

\[ \frac{A(x)}{\text{OPT}(x)} \leq 2 \]

\[ \frac{\text{OPT}(x)}{A(x)} \geq \frac{1}{2} \]

For this problem:

\[ \text{OPT}(x) \leq A(x) \]
Vetex Cover
NP-Hard.
Shall we try greedy again?
How should we be greedy?
Algorithm:

\[\text{GREEDYVERTEXCOVER}(G):\]
\[C \leftarrow \emptyset\]
\[\text{while } G \text{ has at least one edge}\]
\[v \leftarrow \text{vertex in } G \text{ with maximum degree}\]
\[G \leftarrow G \setminus v\]
\[C \leftarrow C \cup v\]
\[\text{return } C\]

Question: Is this ever optimal?
Q: Is it a 2-approx?

A: No:

Remove the blue vertex... And add it to the VC

Remove red vertex

OPT:

Greedy:
Thm: Greedy VC is an $O(\log n)$ approximation:

$$\text{Greedy} \leq O(\log n) \cdot \text{OPT}$$

pf: Let $G_i = \text{graph in } i^{th}$ iteration.

Let $d_i = \text{max degree in } G_i$

GreedyVertexCover($G$):

1. $C \leftarrow \emptyset$
2. $G_0 \leftarrow G$
3. $i \leftarrow 0$
4. while $G_i$ has at least one edge
   5. $i \leftarrow i + 1$
   6. $v_i \leftarrow \text{vertex in } G_{i-1} \text{ with maximum degree}$
   7. $d_i \leftarrow deg_{G_{i-1}}(v_i)$
   8. $G_i \leftarrow G_{i-1} \setminus v_i$
   9. $C \leftarrow C \cup v_i$
10. return $C$