a.) CHOP is NP-complete. We can map CHOP efficiently to SPINDLE, and SPINDLE efficiently to MUTILATE. MUTILATE is in NP. What do we know about SPINDLE and MUTILATE?

CHOP, SPINDLE, MUTILATE NP-Complete

b.) We can efficiently map TWEEDLE-DEE to TWEEDLE-DUMB, and TWEEDLE-DUMB is in NP. What do we know about TWEEDLE-DEE and TWEEDLE-DUMB?

T-DEE: Map to T-Dumb O(n)
Solve T-Dumb O(k) if you have some
T-DEE is NP

We don't know anything.

c.) We can efficiently transform YIN into the NP-hard problem YANG. What do we know about YIN?

YIN ≤ YANG, YANG is hard

d.) JOHN and PAUL are NP-complete problems, and RINGO is in P. We can efficiently transform JOHN into RINGO. What do we know about JOHN, PAUL, and RINGO?

John is NP-hard
John alg: map to RINGO poly-time
Solve RINGO poly-time
John is in P
John, Paul, RINGO, and all NP problems are in P

e.) JABBERWOCKIE is NP-complete. We can efficiently map JABBERWOCKIE into JUBJUB. What do we know about JUBJUB?

Jubjub is NP-hard

P
NP
NP-hard
P is in NP
NP-C is NP + NP-hard
Performance of red depends on bucket data structure

Performance (ignoring red): \( O(\text{nodes}) \)

```plaintext

```
Array length = (Max # of node terminals)² + 1

change max_gain
If any node gain < max_gain
adjust node gains for nodes attached to that net
For each net attached to that node
Remove moved node
Update nodes:
Stop if max_gain drops off bottom
max_gain--
While max_gain entry is empty
Find best node:
max_gain points to highest occupied list
Array of lists, organized by gain
(max # of terminals * 2) + 1 = \frac{(\text{# of times each net could change gain})}{\text{# of terminals}} + \text{(array length + moves upwards (in update nodes))}

Operations = Array length + moves upwards (in update nodes)

While max gain array is empty, max gain:

Find best node:

Operations = \frac{(\text{# of times each net could change gain})}{\text{# of terminals/net}}

Adjust node gains

For each attached net:

Update nodes:

Bucket Data Structure Delay
Bucket Data Structure Delay (cont.)

States:
- Frozen - can't be cut by 1 move
- Unfrozen - can't be cut by 1 move
- Cut - can be unfrozen by 1 move
- Stuck - can't be unfrozen by 1 move

Initial States:
- [0, 0^1]
- [0, 1^1]
- [1, 0^1]
- [1, 1^1]
\[ \text{overall} = O(\text{nodes} + \text{terminals}^2) = O(\text{terminals}) \]

Update nodes = \[4 \times \text{terminals} \times (\text{max-terminals} + 1) + 4 \times \text{terminals}\]

Find best node = \text{terminals} * \text{nodes} * \text{nodes}

\text{nodes} > \text{terminals}

\text{end while}:

unlock \text{all nodes};

\text{backtrack}:

\text{end while}:

\text{connected to these nets:}

update nets connected to moved nodes, and nodes

lock moved node:

pick best of two nodes

\text{Find best node from each partition:}

\text{while valid moves}

\text{for # of iterations (4 to 5 iterations)}

\text{Create initial partitioning:}

Final Performance of FM
Partition with clusters, then remove clustering & repeat.

Example: Recursively cluster with neighbor to whom it shares the most edges. Clusters are nodes that probably should be in same partition. Group nodes together into clusters.
Use multiple runs to improve expected value, reduce variance.

Start from multiple random initializations.

Multiple Runs

72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93