Intractability

Reading: Kleinberg & Tardos Ch. 8

What could be going on?

You’ve been asked to solve a *problem* by writing a *program*.

You:

1. come up with your solution
2. implement it
3. cross your fingers while you execute the code

No errors print out, but it also doesn't complete execution, even after a couple minutes!

What could be going on?

What could be going on?

You've been asked to solve a *problem* by writing a *program*.

You:

1. come up with your solution
2. implement it
3. cross your fingers while you execute the code

No errors print out, but it also doesn't complete execution, even after a couple minutes!

Your program might have an infinite loop or recursion or it might just need some more time running...

Complexity CardLine

Building intuition

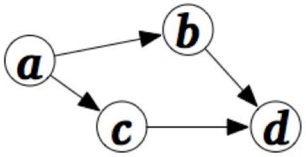
EmptyList
Given a list, determine if it has no elements.

EmptyList(()) → true
EmptyList((a,b)) → false

SubsetSum
Given a list of integers and target sum, determine if a subset adds to the target.

SubsetSum((1, 3, 4), 5) → true
SubsetSum((1, 3, 4), 6) → false

SimplePath
Given a graph and vertices *s* and *t*, determine if there is a path from *s* to *t* that does not visit any vertex more than once.
SimplePath(a, d) → true
SimplePath(b, c) → false



Complexity classes

- A problem belongs to *complexity class* **TIME(T(n))** if there exists an O(T(n)) time algorithm that solves it
  - EmptyList $\in$ TIME(1)
  - SimplePath $\in$ TIME(n)
  - SUBSET-SUM $\in$ TIME(2^n)

Note: here, "abuse" notation
n is complete input size (edges + vertices)

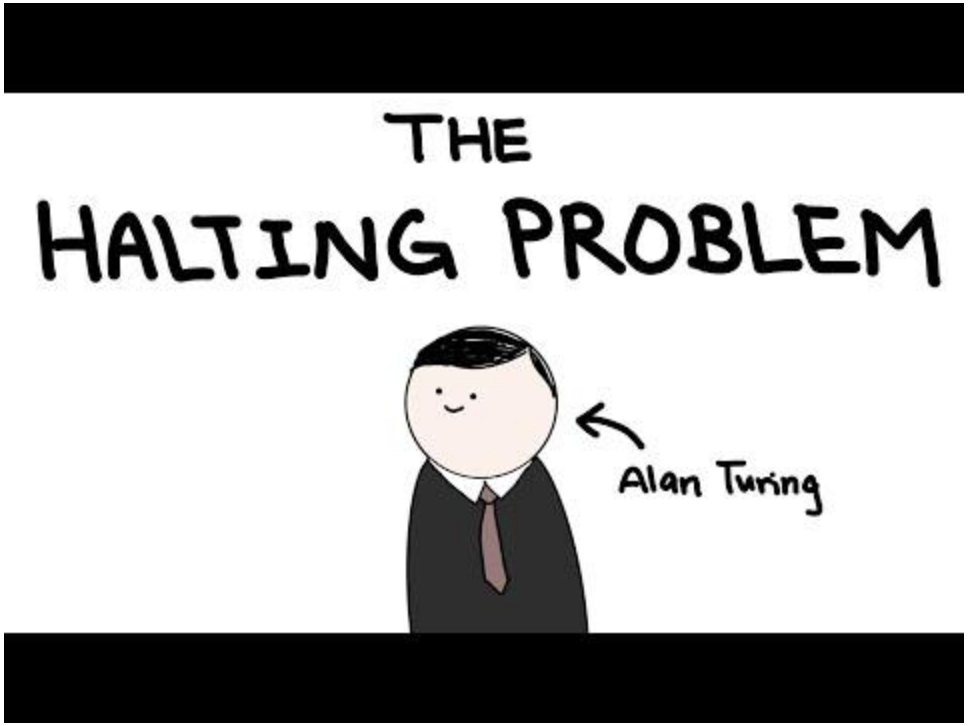
- $\mathbf{TIME(1)} \subseteq \mathbf{TIME(}\ \mathbf{log}\ \boldsymbol{n}\ \mathbf{)} \subseteq \mathbf{TIME(}\ \boldsymbol{n}\ \mathbf{)} \subseteq \mathbf{TIME(}\ \boldsymbol{n}\ \mathbf{log}\ \boldsymbol{n}\ \mathbf{)}$
  $\subseteq \mathbf{TIME(}\ \boldsymbol{n}^{\mathbf{2}}\ \mathbf{)} \subseteq \mathbf{\ldots\ TIME(}\ \boldsymbol{n}^{\boldsymbol{k}}\ \mathbf{)}$
  $\subseteq$ TIME(2^n) …

How hard can a problem be?

Unsolvable!

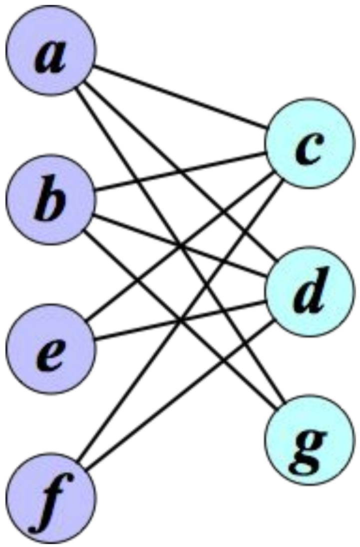
The Halting Problem

How hard can a problem be?



Perfect matchings for bipartite graphs

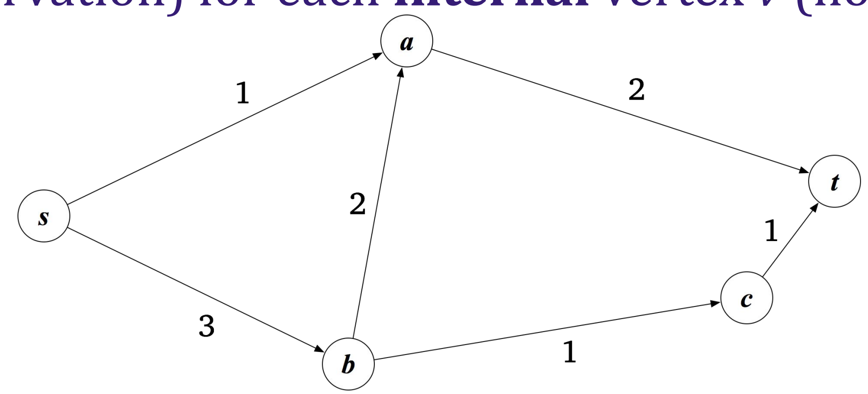
- Bipartite graph $G = (V, E)$
- V can be partitioned into disjoint vertex sets: X and Y
- Edges: one endpoint in each set



- **Matching**: set M of family-animal pairs each family/animal participates in *at most* one pair
- **Perfect matching**: assuming n families and n animals, each family/animal in *exactly* one pair

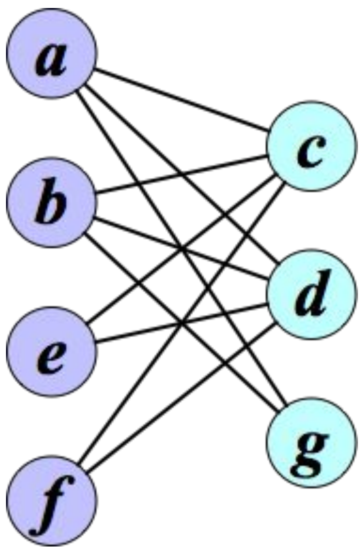
Maximum flow in a network

- Input:
  - Directed graph G = (V,E) with edge capacities c_e for each edge e
  - Source vertex s
  - Sink vertex t
- Find maximum “flow” from source to target
  - ***s*-*t* flow** $f : E \rightarrow \mathbf{R}^+$ such that
    - (capacity) $0 \leq f(e) \leq c_e$ for each edge e
    - (conservation) for each **internal** vertex v (not s or t)

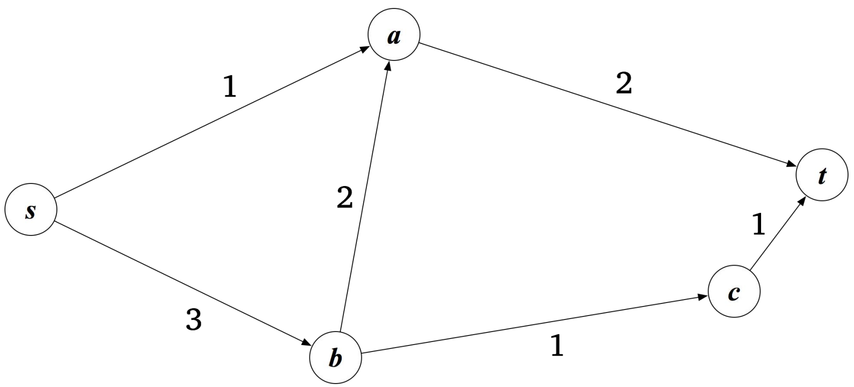


Which problem is harder?

Perfect matching in bipartite graphs



Maximum flow in networks



Can we compare problems?

- Claim: bipartite matching is "no harder than" network flow
- Bipartite matching ≤ Network flow
  - *Reduce* the problem of bipartite matching to the problem of network flow
  - How long does this take?
- Edmonds-Karp specialization of Ford-Fulkerson Network Flow: O(VE^2)
  - *Network Flow is polynomial time*
- Then *Bipartite matching is polynomial time*

Reductions

- *Reduction* from problem A to problem B
  - function f such that $x \in A$ if and only if $f(x) \in B$
  - f converts "yes" instance of A to "yes" instance $f(x)$ of B, and
  - f converts "no" instance of A to "no" instance $f(x)$ of B
  - $A \leq B$ means "A is reducible to B" if reduction exists
- $A \leq$p B
  - "*reducible* in polynomial time"
  - if there is reduction that can be computed in poly. time

P and NP

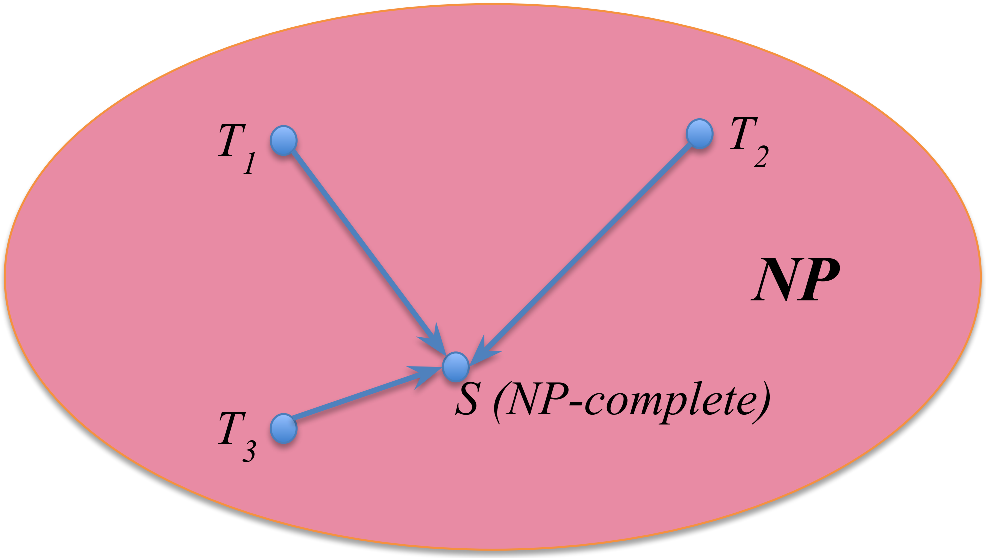
- Problem belongs to *complexity class* **TIME(T(n))** if there exists an O(T(n)) time algorithm that solves it
- **TIME(1) ⊆ TIME(log n) ⊆ TIME(n) ⊆ TIME(n log n) ⊆ TIME(n^2) ⊆ … TIME(n^k)** ⊆ TIME(2^n) …
- **P** = class of problems for which poly. time algorithm exists
- **NP** = class of problems that are *verifiable* in polynomial time

P and NP

- P: class of problems for which poly. time algorithm exists
- NP: class of problems verifiable in poly. time
  - examples: IsSorted, Hamiltonian path, Subset-Sum
  - equivalent to class of problems for which a *nondeterministic* polynomial time algorithm exists
    - *nondeterministic*: power of “lucky” guessing
      - can make guesses in the algorithm
      - if correct guess exists, guaranteed to choose it
- P $\subseteq$ NP

NP-completeness

- Problem S is *NP-hard* if:
  - $T \leq_p S$ for all $T \in NP$
- Problem S is *NP-complete* if:
  - $S \in NP$
  - $T \leq_p S$ for all $T \in NP$



NP-complete problems

- NP's "hardest problems"
- 3SAT: boolean formulas for which satisfying truth assignment exists
  - conjunctive normal form with 3 literals per clause
    - (a OR b OR c) AND (d OR !e OR f) AND …
  - literal: variable or its negation
- 3SAT is NP-complete
  - 3SAT $\in$ NP
  - T $\leq$p 3SAT for all T $\in$ NP *[via Cook-Levin Theorem]*

How hard is Subset-Sum?

k: # of literals per clause

- Show 3SAT $\leq$p SUBSET-SUM:
  - Given boolean formula in 3CNF form with n variables and m clauses
  - Create a set of 2n + (k-1)m decimal numbers
    - each number has n + k digits
    - table schematic: rows are numbers, columns are digits
    - two types of columns
      - first n columns labeled by variables
      - last m columns labeled by clauses
    - two types of rows
      - 2n variable value rows
        - one row for variable x and one for negation !x
          - variable columns: 1 if column for x, 0 otherwise
          - clauses columns: 1 in column if clause contains literal, 0 otherwise
      - 2m *[generally (k-1)m]* "slack" rows
        - 2 rows for each clause c
          - variable columns: 0
          - clauses columns: 1 if column for c, 0 otherwise
  - target value is decimal number: n 1's, followed by m 3's
    - sum to 1 in variable columns <=> each variable assigned true or false
    - sum to 3 in clause columns <=> at least one literal certify truth
      - (may need to pad with slack variables)

Example: B = (x or y or z) and (!x or y or !z)

```
        x   y   z   c1  c2
 x:     1   0   0   1   0          // x = T
!x:     1   0   0   0   1          // x = F
 y:     0   1   0   1   1          // y = T
!y:     0   1   0   0   0          // y = F
 z:     0   0   1   1   0          // z = T
!z:     0   0   1   0   1          // z = F
c1:     0   0   0   1   0          // slack for c1 if <3 true literals
c1:     0   0   0   1   0          // slack for c1 if <3 true literals
c2:     0   0   0   0   1          // slack for c2 if <3 true literals
c2:     0   0   0   0   1          // slack for c2 if <3 true literals
--------------------
t:     1   1   1   3   3
```

- summary: B reduces to { 10010, 10001, 1011, 1000, 101, 10, 10, 1, 1 }, target value 11133
  - solution to Subset-sum instance gives: {10, 10, 101, 1011, 10001}
  - corresponding to: both c1 slack rows, !z, y and !x
    - truth assignment is: x - F, y - T, z - F
    - clause 1 only has one true literal (y), so needs both slack rows to get to 3
    - clause 2 has all three literals (!x, y, !z), so uses no slack rows

Does P = NP?

solve *any* NP-complete problem in polynomial time