Balanced Programming

Group Static

Shanghai Jiao Tong University

December 26, 2016

Introduction

- Very useful idea when confronting the following cases:

Introduction

- Very useful idea when confronting the following cases:
  1. There are two algorithms, both of which have different features, such as one has lower complexity in query and the other has lower complexity in modification.

Introduction

- Very useful idea when confronting the following cases:
  1. There are two algorithms, both of which have different features, such as one has lower complexity in query and the other has lower complexity in modification.
  2. The whole problem can be divided into several problems called “big problems” and these problems can be also divided into several smaller but common problems called “small problems”.

Introduction

- Very useful idea when confronting the following cases:
  1. There are two algorithms, both of which have different features, such as one has lower complexity in query and the other has lower complexity in modification.
  2. The whole problem can be divided into several problems called "big problems" and these problems can be also divided into several smaller but common problems called "small problems".
- The main idea is these algorithms learning from each other.

Problem

Find a smallest non-negative number x satisfying

$$A^x = B \ (mod\ P)$$

which P is a prime, $A, B \in [0, P)$.

Naive Approach

According to Fermat Theory, when A, P is coprime, we have:

$$A^P = A \ (mod\ P)$$

the only special case is $A = 0$, and in this case, B must be zero.

For $A \neq 0$, we can just iterate x from 0 to $P - 1$ checking if $A^x = B \ (mod\ P)$. the complexity is $O(P)$.

Yet Another Naive Algorithm

We mapped $A^x \to x, x \in [0, P-1)$ by using a Hash-Table.

For finding B, we just need to query B in this Hash-Table, which only requires $O(1)$ time.

Preparing the Hash-Table needs $O(P)$ time, which the total complexity is $O(P+1) = O(P)$.

Balanced Programming

- First we choose a number $S \in [1, P-1]$.
- We mapped $A^x \to x, x \in [0, S)$ by using a Hash-Table.
- Calculate A^{-S} by using Fast Exponentiation.

In this step, we needs $S + \log P$ operations.

Balanced Programming

x can be represented as $i \times S + j$, we can transform the equation:

$$A^{i \times S + j} = B \Leftrightarrow A^j = B \times (A^{-S})^i \ (mod\ P)$$

which $j \in [0, S)$, and $i \leq \frac{P}{S}$.

We can just iterate i from 0 to $\frac{P}{S}$ checking if $B \times (A^{-S})^i$ is in Hash-Table. In worst occasion, we need to iterator $\frac{P}{S} + 1$ times.

Total complexity evaluation

As you can see, the total operations we need to do is

$$S + \log P + \frac{P}{S} + 1$$

we want to choose S optimally to have the best total complexity.

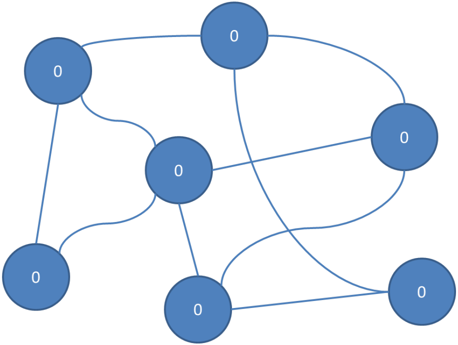
when $S = \sqrt{P}$, the total operations is:

$$S + \log P + \frac{P}{S} + 1 = 2\sqrt{P} + \log P + 1 = O(\sqrt{P})$$

thus we get a $O(\sqrt{P})$ algorithm by choosing S optimally.

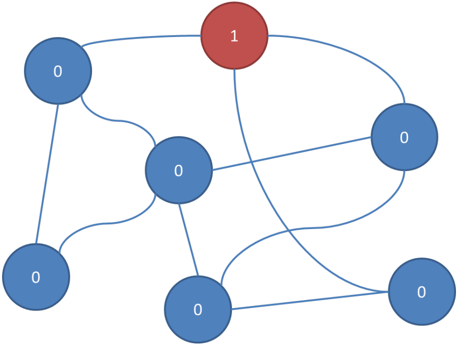
Another Problem

- an undirected graph



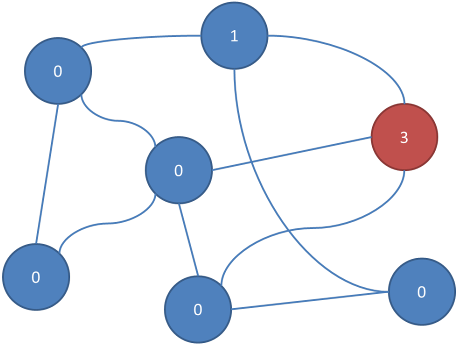
Another Problem

- an undirected graph
- add x to u



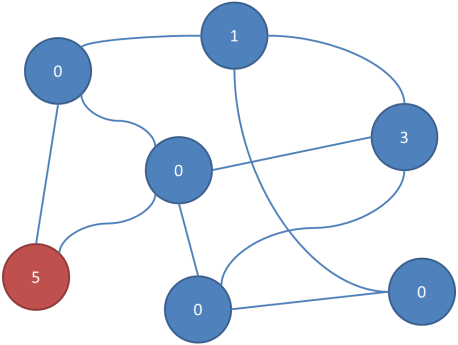
Another Problem

- an undirected graph
- add x to u



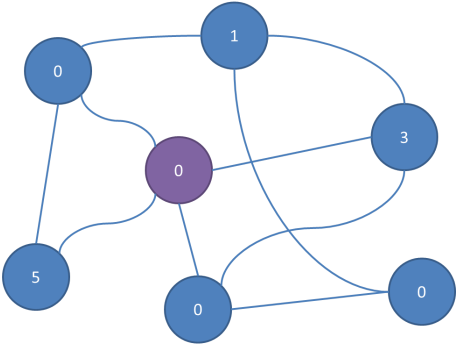
Another Problem

- an undirected graph
- add x to u



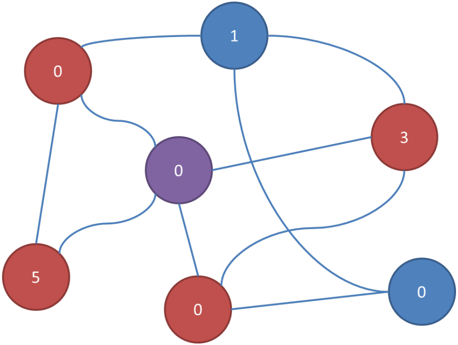
Another Problem

- an undirected graph
- add x to u
- query neighbors' sum



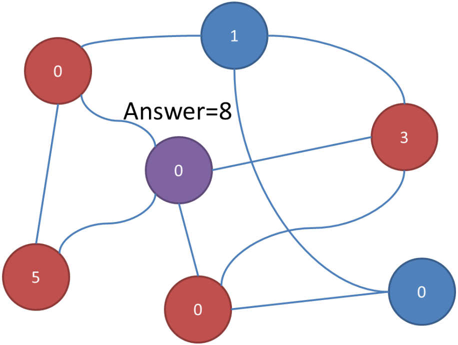
Another Problem

- an undirected graph
- add x to u
- query neighbors' sum



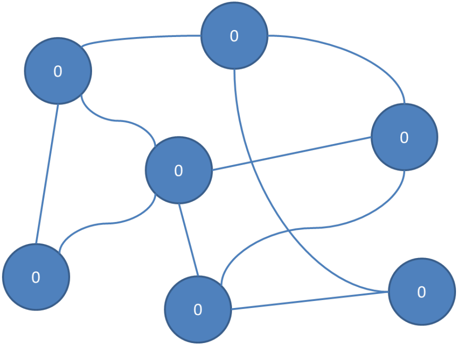
Another Problem

- an undirected graph
- add x to u
- query neighbors' sum
- $O(m) = n$



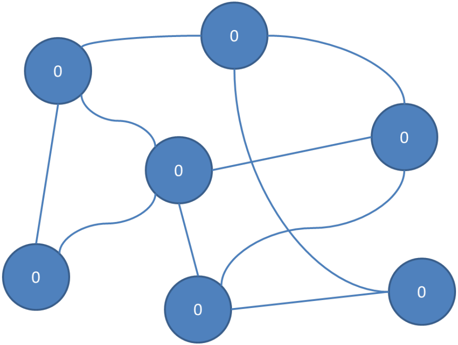
Naive Approach 1

- store the value $v[x]$



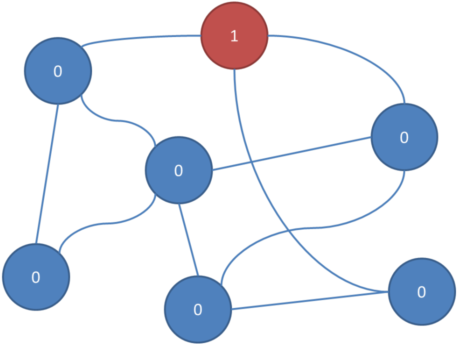
Naive Approach 1

- store the value $v[x]$
- $O(n)$ space



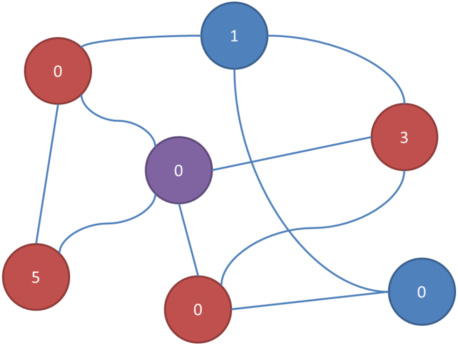
Naive Approach 1

- store the value $v[x]$
- $O(n)$ space
- $O(1)$ time for add



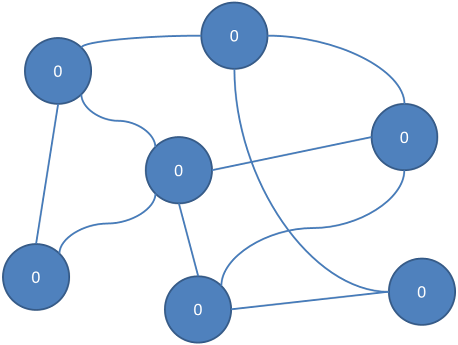
Naive Approach 1

- store the value $v[x]$
- $O(n)$ space
- $O(1)$ time for add
- $O(m)$ time for query



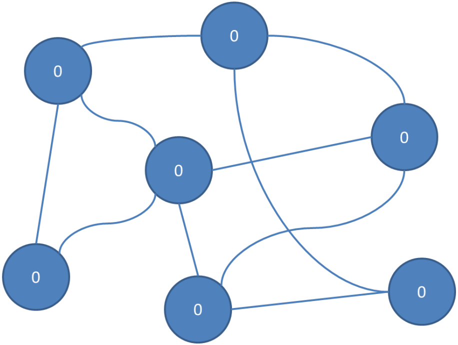
Naive Approach 2

- store the sum $s[x]$



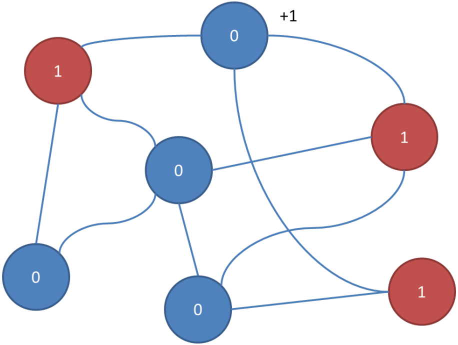
Naive Approach 2

- store the sum $s[x]$
- $O(n)$ space



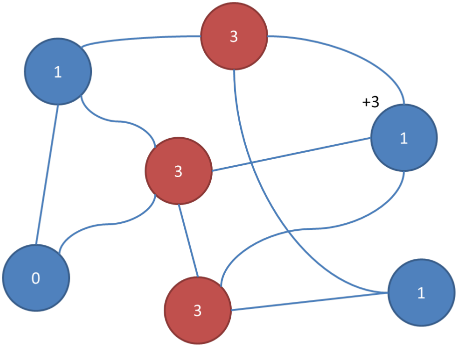
Naive Approach 2

- store the sum $s[x]$
- $O(n)$ space
- $O(n)$ time for add



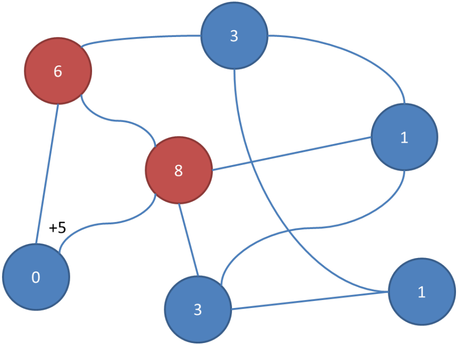
Naive Approach 2

- store the sum $s[x]$
- $O(n)$ space
- $O(n)$ time for add



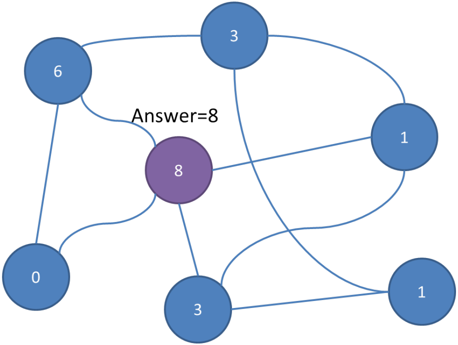
Naive Approach 2

- store the sum $s[x]$
- $O(n)$ space
- $O(n)$ time for add



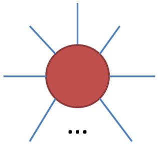
Naive Approach 2

- store the sum $s[x]$
- $O(n)$ space
- $O(n)$ time for add
- $O(1)$ time for query



Observation

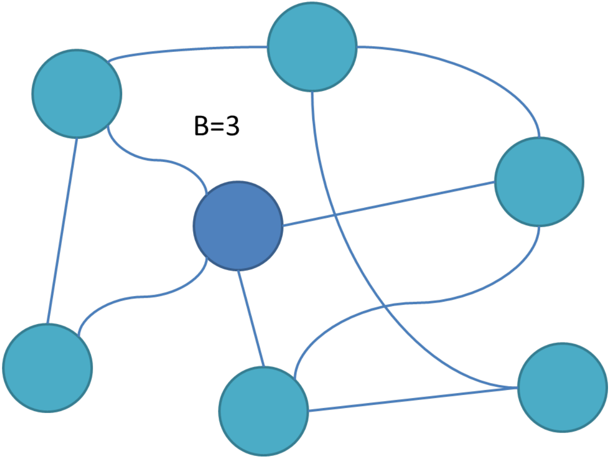
- bad when large $deg(x)$



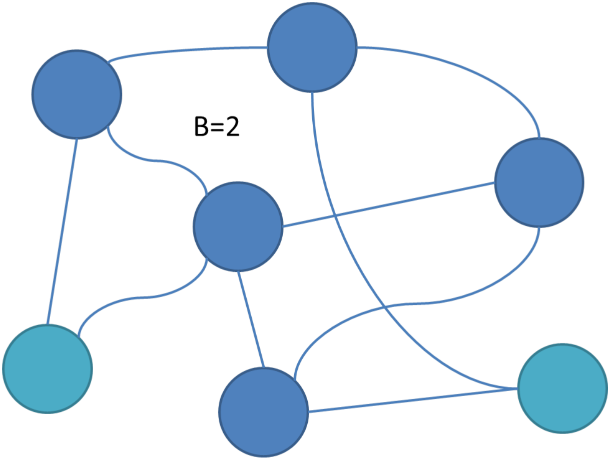
Observation

- bad when large $deg(x)$
- "heavy" when $deg(x) > B$

Heavy-Light Divide

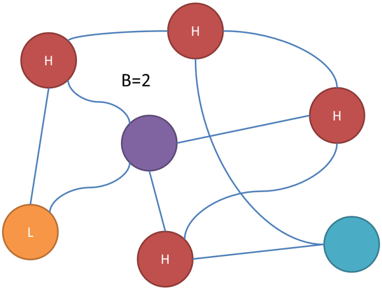


Heavy-Light Divide



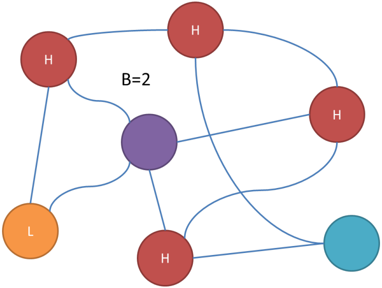
Combined Approach

- $s[x] = \sum v[heavy\ neighbors] + \sum v[light\ neighbors]$



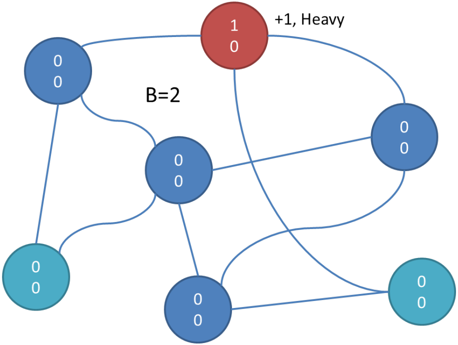
Combined Approach

- $s[x] = \sum v[\textit{heavy neighbors}] + \sum v[\textit{light neighbors}]$
- for $\sum v[\textit{heavy neighbors}]$ use approach 1
- for $\sum v[\textit{light neighbors}]$ use approach 2



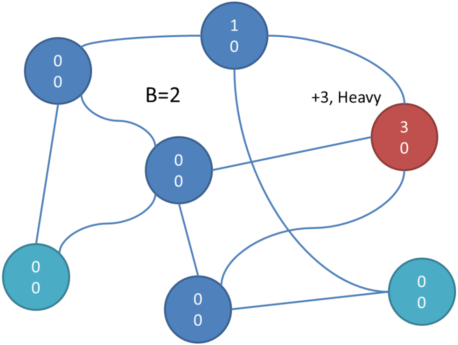
$Add(u, x)$ details

- if v is heavy, $vh[u] \leftarrow vh[u] + x$



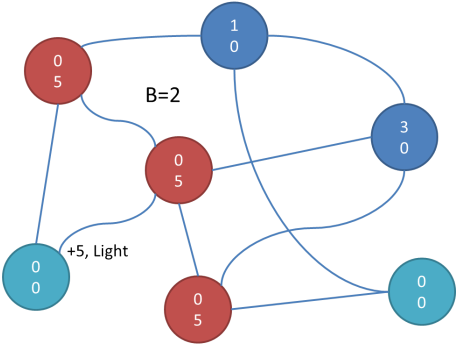
$Add(u, x)$ details

- if v is heavy,
  $vh[u] \leftarrow vh[u] + x$



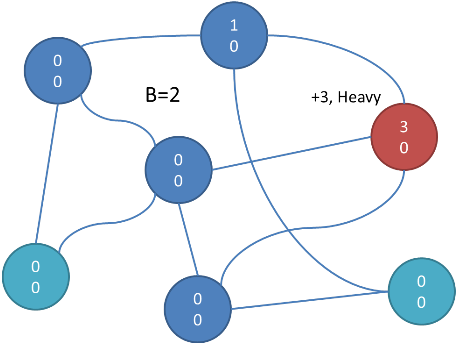
$Add(u, x)$ details

- if v is heavy,
  $vh[u] \leftarrow vh[u] + x$
- if u is light,
  $sl[v] \leftarrow sl[v] + x$,
  u, v are neighbors



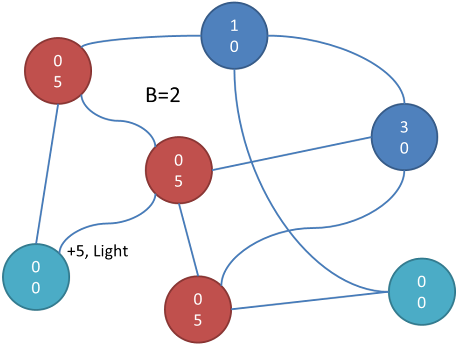
$Add(u, x)$ details

- if v is heavy,
  $vh[u] \leftarrow vh[u] + x$
- if u is light,
  $sl[v] \leftarrow sl[v] + x$,
  u, v are neighbors
- $O(1)$ time for heavy



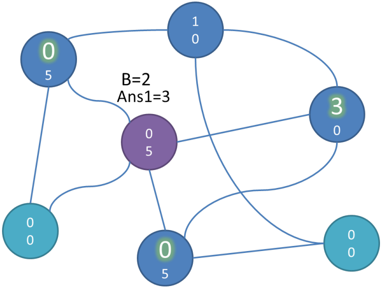
$Add(u, x)$ details

- if v is heavy,
  $vh[u] \leftarrow vh[u] + x$
- if u is light,
  $sl[v] \leftarrow sl[v] + x$,
  u, v are neighbors
- $O(1)$ time for heavy
- $O(B)$ time for light



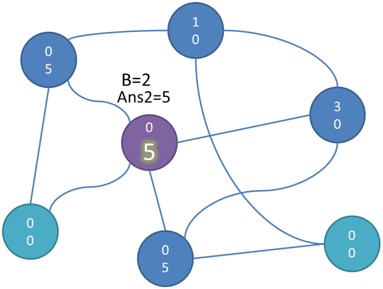
$Query(x)$ details

- $\sum v[heavy\ neighbors] = \sum_{y\ is\ heavy\ neighbor} vh[y]$



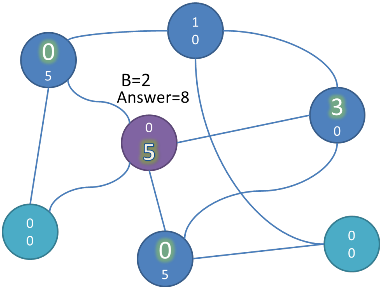
$Query(x)$ details

- $\sum v[heavy\ neighbors] = \sum_{y\ is\ heavy\ neighbor} vh[y]$
- $\sum v[light\ neighbors] = sl[x]$



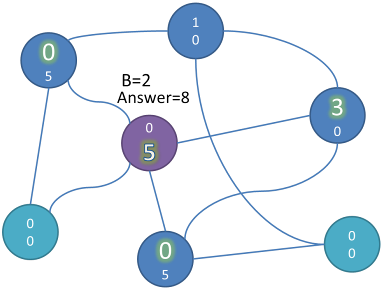
$Query(x)$ details

- $\sum v[heavy\ neighbors] = \sum_{y\ is\ heavy\ neighbor} vh[y]$
- $\sum v[light\ neighbors] = sl[x]$



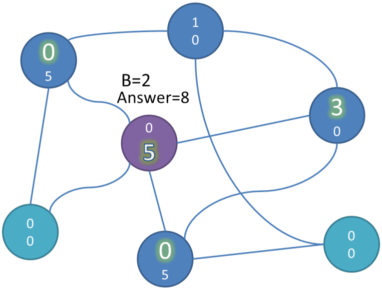
$Query(x)$ details

- $\sum v[heavy\ neighbors] = \sum_{y\ is\ heavy\ neighbor} vh[y]$
- $\sum v[light\ neighbors] = sl[x]$
- $O(1 + cnt\_heavy)$ time



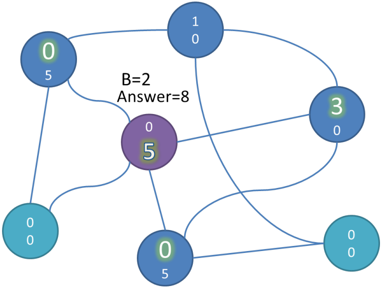
$Query(x)$ details

- $\sum v[heavy\ neighbors] = \sum_{y\ is\ heavy\ neighbor} vh[y]$
- $\sum v[light\ neighbors] = sl[x]$
- $O(1 + cnt\_heavy)$ time
- $cnt\_heavy \cdot B \leq 2m = O(n)$



$Query(x)$ details

- $\sum v[heavy\ neighbors] = \sum_{y \text{ is heavy neighbor}} vh[y]$
- $\sum v[light\ neighbors] = sl[x]$
- $O(1 + cnt\_heavy)$ time
- $cnt\_heavy \cdot B \leq 2m = O(n)$
- $O(n/B)$ time



Balance

- $O(n)$ space

Balance

- $O(n)$ space
- $O(B)$ time for add

Balance

- $O(n)$ space
- $O(B)$ time for add
- $O(n/B)$ time for query

Balance

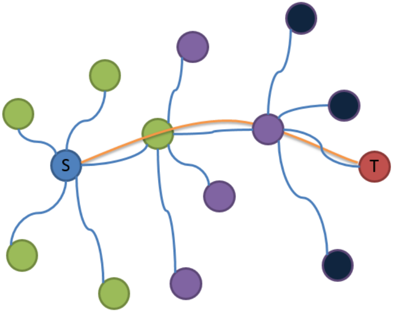
- $O(n)$ space
- $O(B)$ time for add
- $O(n/B)$ time for query
- make $B = \sqrt{n}$

Balance

- $O(n)$ space
- $O(B)$ time for add
- $O(n/B)$ time for query
- make $B = \sqrt{n}$
- $O(\sqrt{n})$ for each operation

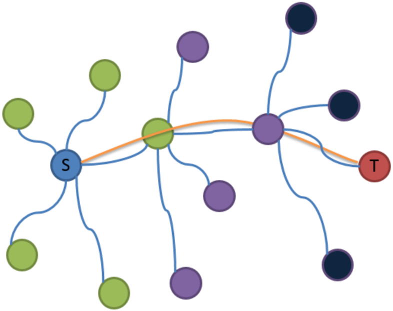
Meet in the middle

- Given an undirected graph whose nodes' degree is 5 and find a path from S to T.



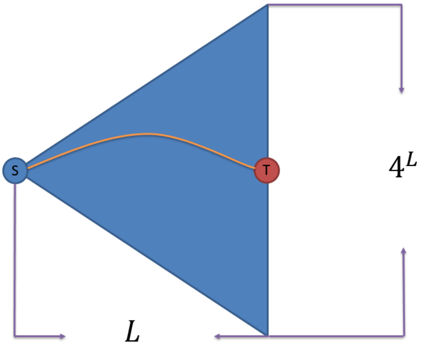
Meet in the middle

- Given an undirected graph whose nodes' degree is 5 and find a path from S to T.
- $\text{dist(S, T)} = \text{L}$



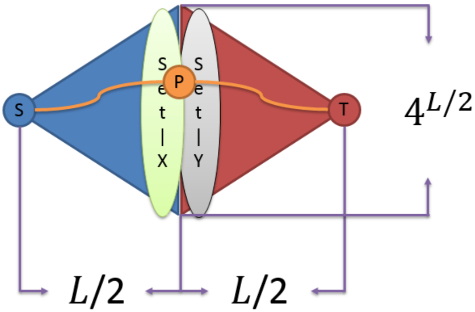
Meet in the middle

- Given an undirected graph whose nodes' degree is 5 and find a path from S to T.
- dist(S, T) $=$ L
- $O(4^L)$



Meet in the middle

- Given an undirected graph whose nodes' degree is 5 and find a path from S to T.
- dist(S, T) = L
- $O(4^L)$
- Meet in the middle
- $O(4^{L/2})$

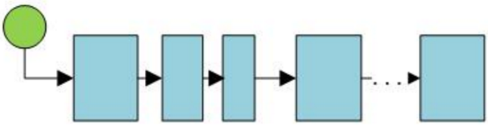


Some other applications

- Mo's algorithm

Some other applications

- Mo's algorithm
- Bolcked list



Some other applications

- Mo's algorithm
- Bolcked list
- Dinic(capacity is 1)

Some other applications

- Mo's algorithm
- Bolcked list
- Dinic(capacity is 1)
- ......

Thanks

Thanks for listening.
Questions are welcomed.