Balanced Programming

Group Static

Shanghai Jiao Tong University

December 27, 2016

Overview

Features

- Kind of a useful designing idea.

Features

- Kind of a useful designing idea.
- Easy implementation and good performance in practice.

Balances

- Balanced in learning from each each other.

Balances

- Balanced in learning from each each other.
- Balanced in the problem-solving processes.

Available occiasion

- There are two algorithms, both of which have different features, such as one can answer queries very quickly and the other cam handle modification swiftly.

Available occiasion

- There are two algorithms, both of which have different features, such as one can answer queries very quickly and the other cam handle modification swiftly.
- There is an algorithm to solve the problems, but is quite complicated. Actually this algorithm is not the best choice in practice. The idea of balanced programming might be used to produced a suitable algorithm.

Overview

Problem

Find a smallest non-negative number x satisfying

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

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

Naive Approach

According to Fermat Theorem, when A, P are coprime, we have:

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

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 $P - 1$ to check if $A^x \equiv B \ (mod \ P)$. The complexity is $O(P)$.

Yet Another Naive Algorithm

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

In order to find B, we just need to query B in this Hash-Table, which only takes $O(1)$ time.

Precalculating this Hash-Table costs $O(P)$ time, and the total complexity of which 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

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

x can be represented as $i \times S + j$, we can do such transforming:

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

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

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

Total complexity evaluation

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

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

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

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

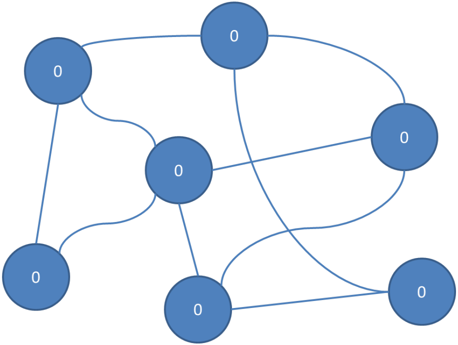
$$S + \log P + \frac{P}{S} = 2\sqrt{P} + \log P = O(\sqrt{P})$$

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

Overview

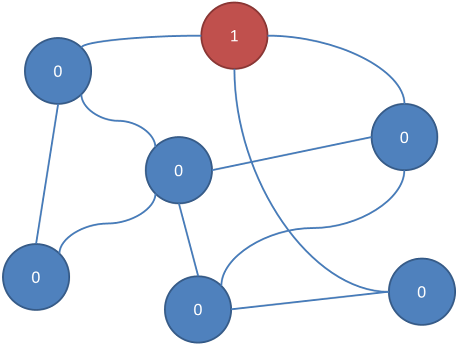
Another Problem

- an undirected graph



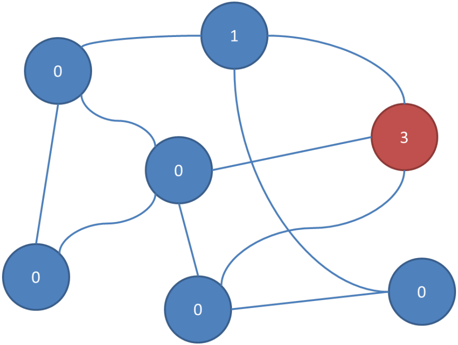
Another Problem

- an undirected graph
- add x to u



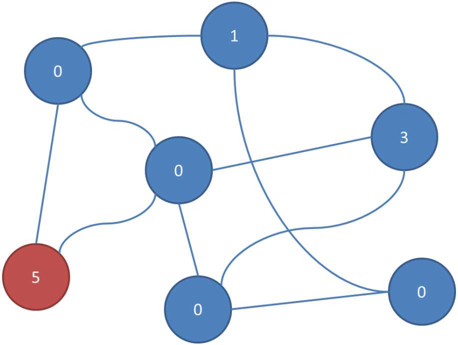
Another Problem

- an undirected graph
- add x to u



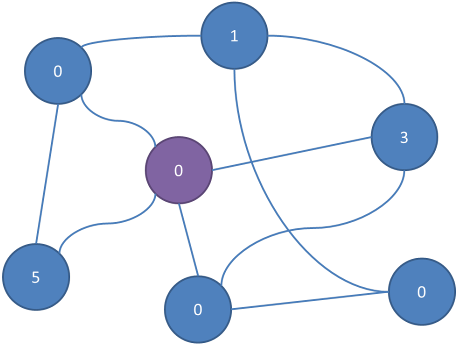
Another Problem

- an undirected graph
- add x to u



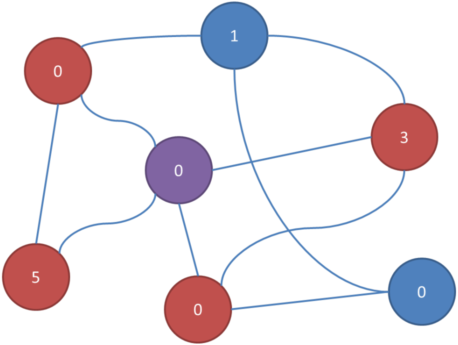
Another Problem

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



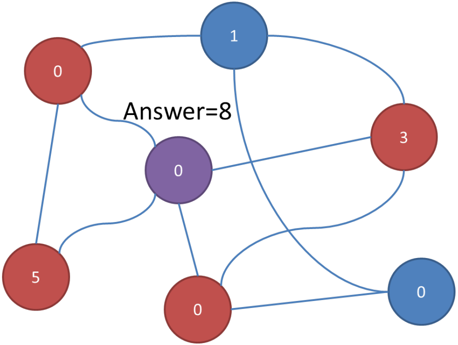
Another Problem

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



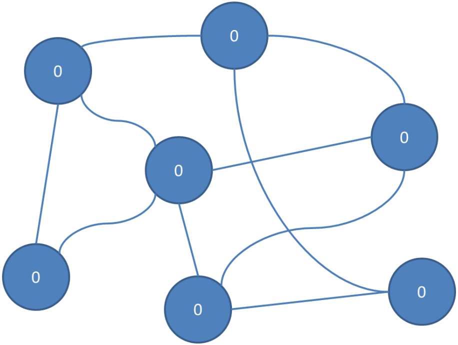
Another Problem

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



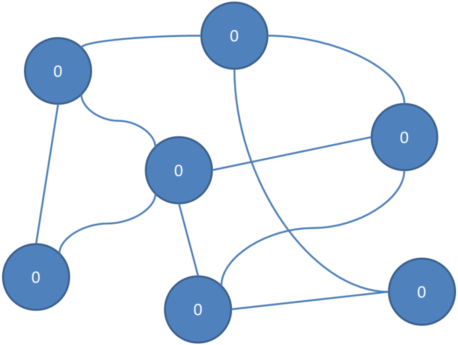
Naive Approach 1

- store the value $v[x]$



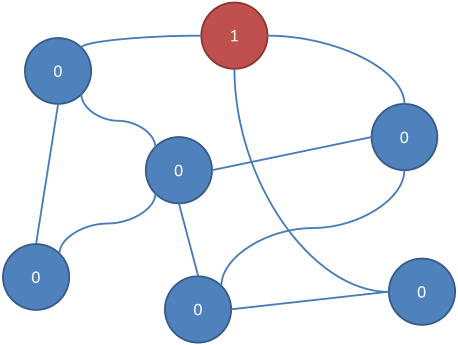
Naive Approach 1

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



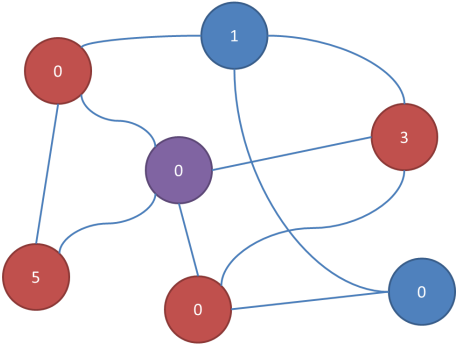
Naive Approach 1

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



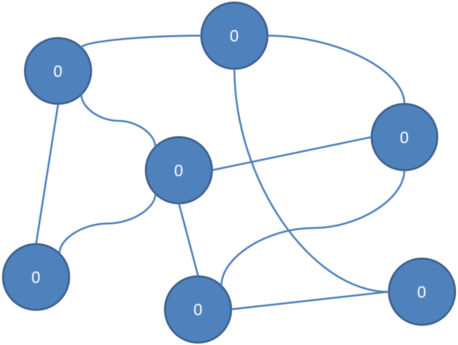
Naive Approach 1

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



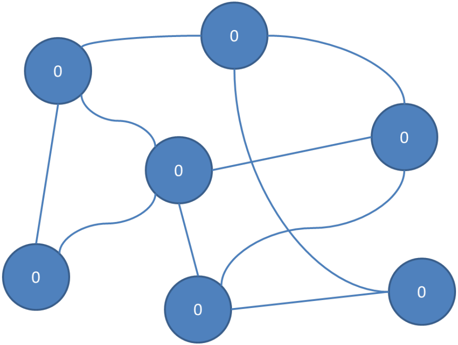
Naive Approach 2

- store the sum $s[x]$



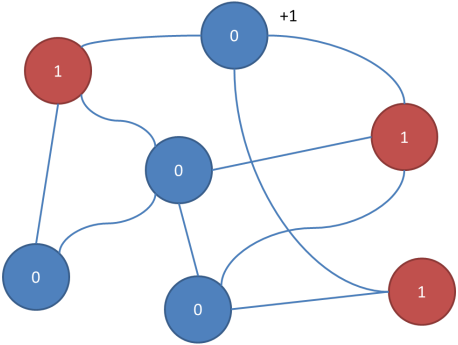
Naive Approach 2

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



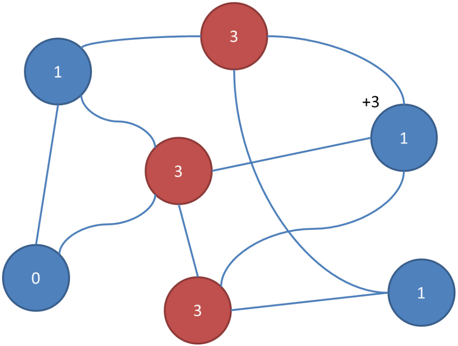
Naive Approach 2

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



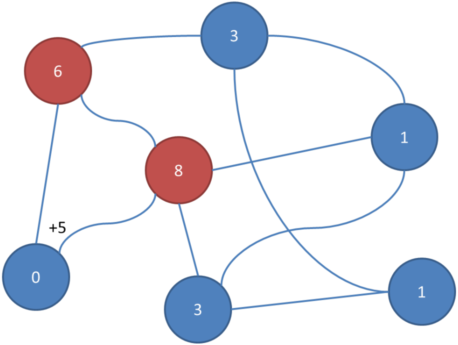
Naive Approach 2

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



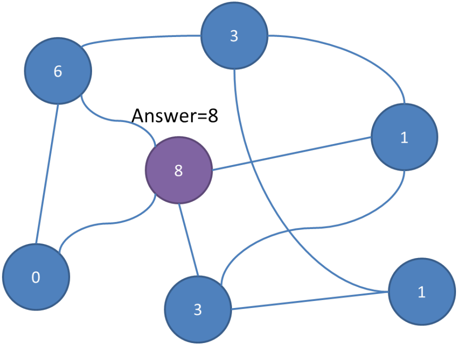
Naive Approach 2

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



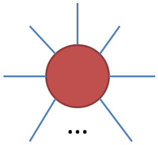
Naive Approach 2

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



Observation

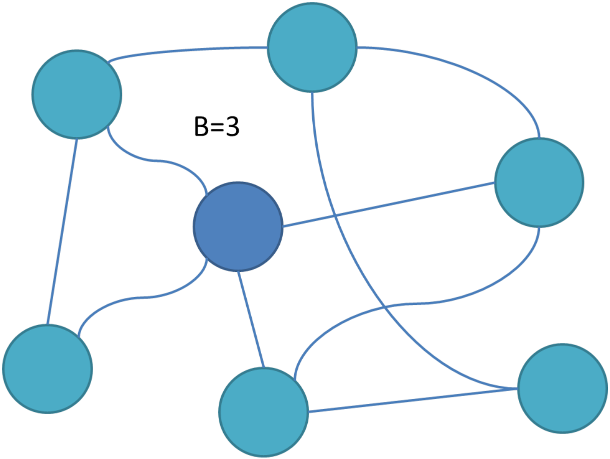
- bad when large $deg(x)$



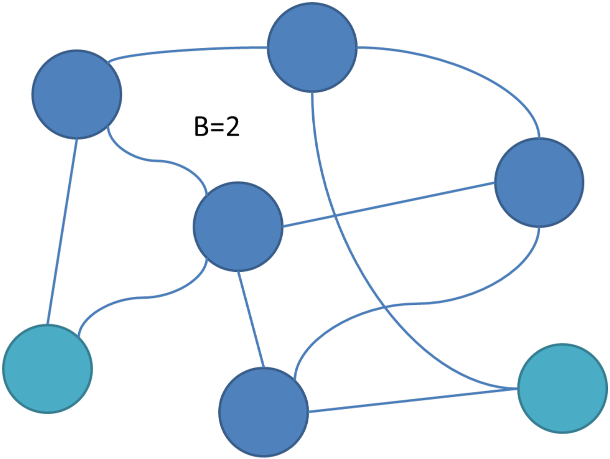
Observation

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

Heavy-Light Divide

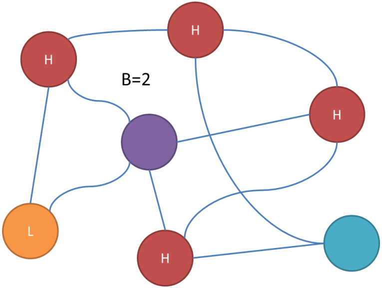


Heavy-Light Divide



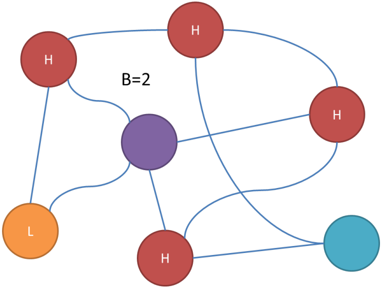
Combined Approach

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



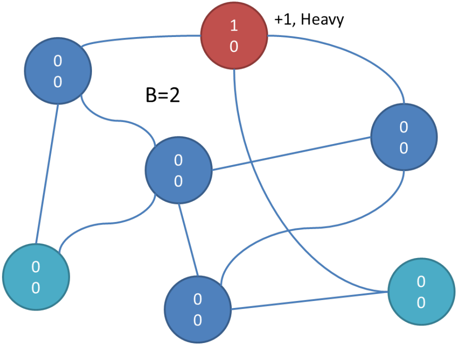
Combined Approach

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



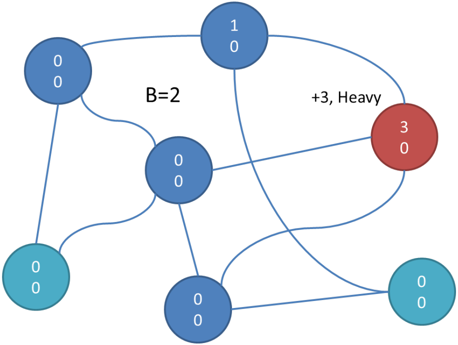
$Add(u, x)$ details

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



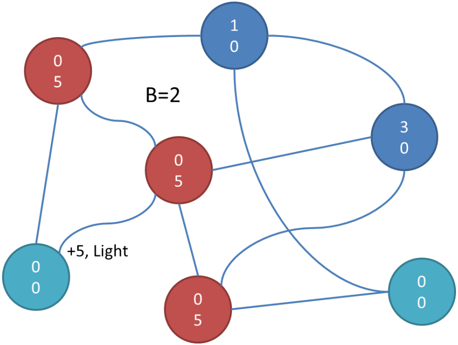
$Add(u, x)$ details

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



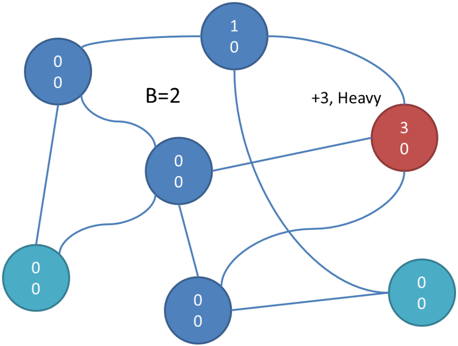
$Add(u, x)$ details

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



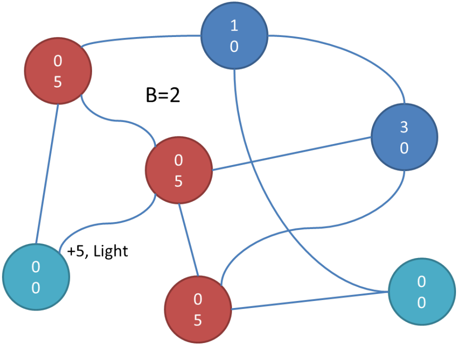
$Add(u, x)$ details

- if u 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



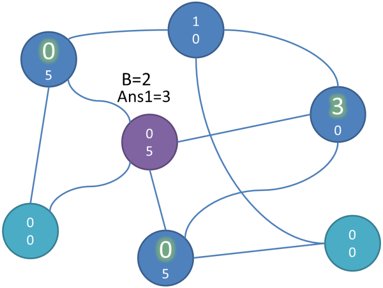
$Add(u, x)$ details

- if u 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



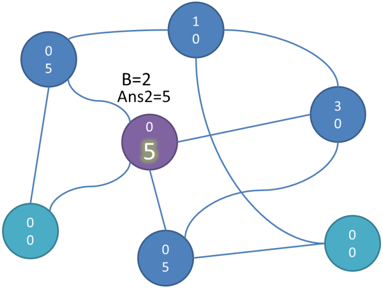
$Query(x)$ details

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



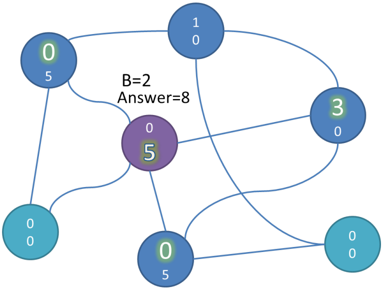
$Query(x)$ details

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



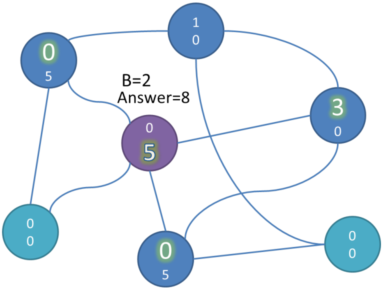
$Query(x)$ details

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



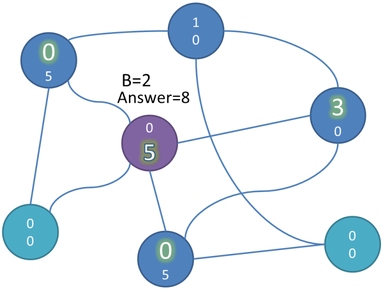
$Query(x)$ details

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



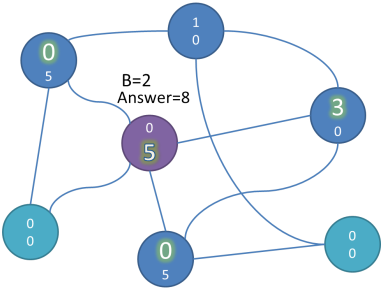
$Query(x)$ details

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



$Query(x)$ details

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

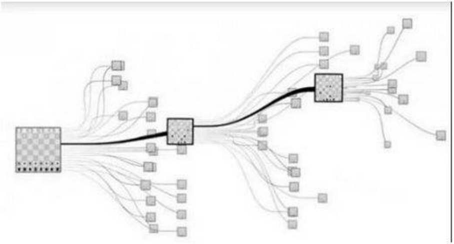
Balance

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

Overview

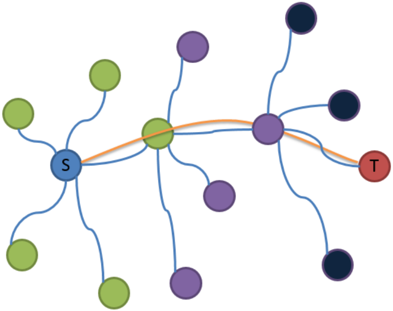
Search algorithm

- Search algorithm
- exponential : $O(x^{depth})$



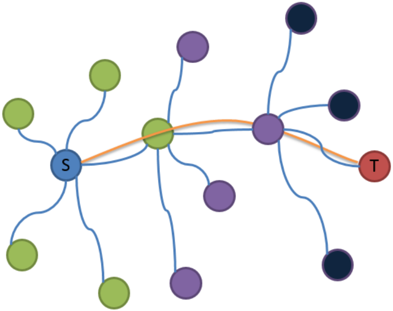
Meet in the middle

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



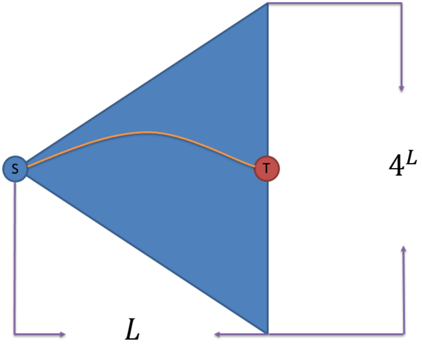
Meet in the middle

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



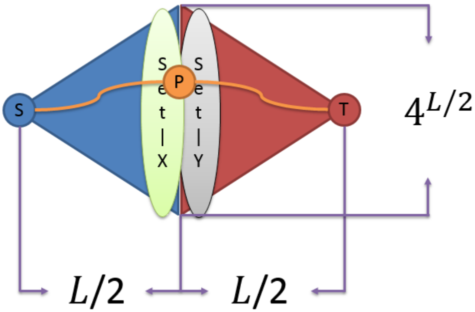
Meet in the middle

- 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
- $O(4^{L/2})$

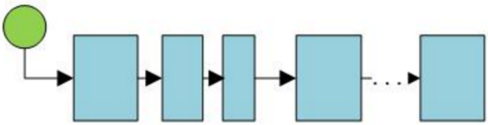


Some other applications

- Mo's algorithm

Some other applications

- Mo's algorithm
- Blocked list



Some other applications

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

Some other applications

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

Thanks

Thanks for listening.
Questions are welcomed.