CS4311
Design and Analysis of Algorithms

Tutorial: An Introduction to Approximation Algorithms

About this Tutorial

- Decision vs Optimization
- NP-Hard Problems
- Dealing with NP-Hard Problems
  - Exact Algorithm
  - Randomized Algorithm
  - Approximation Algorithm (today's focus)

Decision vs Optimization

- Last time, we have talked about decision problems, in which the answer is either YES or NO

  E.g., Peter gives us a map $G = (V,E)$, and he asks us if there is a path from A to B whose length is at most 100

Decision vs Optimization

- A more natural type of problem is called optimization problems, in which we want to obtain a best solution

  E.g., Peter gives us a map G = (V,E), and he asks what is the length of the shortest path from A to B

- Usually, the answer to an optimization problem is a number

Decision vs Optimization

- Two major types of optimization problems: minimization or maximization
  - Previous example is a minimization problem

- An example for a maximization problem:
  - Peter gives us a map $G = (V,E)$, and he asks what is the maximum number of edge-disjoint paths from A to B

Decision vs Optimization

- Decision problem and optimization problem are closely related :

(1) Peter gives us a map $G = (V,E)$, and he asks what is the length of the shortest path from A to B

(2) Peter gives us a map $G = (V,E)$, and he asks us if there is a path from A to B with length at most k

Decision vs Optimization

- We see that if Problem (1) can be solved, we can immediately solve Problem (2)
- In general, if the optimization version can be solved, the corresponding decision version can be solved !
  - What if its decision version is known to be NP-complete ??

Decision vs Optimization

- For example, the following is a famous optimization problem called Max-Clique :

> Given an input graph G, what is the size of the largest clique in G ?

- Its decision version, Clique, is NP-complete:

> Given an input graph G, is there a clique of size at least k ?

NP-Hard

- If the decision version is NP-complete, then it is unlikely that the optimization problem has a polynomial-time algorithm
  - We call such optimization problem an NP-hard problem

- So, perhaps no polynomial-time algorithm may exist... Should we give up solving the NP-hard problems?

Dealing with NP-Hard problems

- Although a problem is NP-hard, it does not mean that it cannot be solved
- At least, we can try naïve brute force search, only that it needs exponential time
- Other common strategies :
  - Exact Algorithm
  - Randomized Algorithm
  - Approximation Algorithm

Exact Algorithm

- Given a graph G with n vertices,
  - a brute force approach to solve the Max-Clique problem is to select every subset of G, and test if it is a clique
  - Running time: $O(2^n n^2)$ time
- Though time is exponential, it works well when n is small, and we can improve it …
- Tarjan & Trojanowski [1977]: $O(1.26^n)$ time

Randomized Algorithm

- Use randomization to help
- Idea 1: Design an algorithm that answers correctly most of the time (but sometimes may give wrong answer), and it always run in polynomial time
- Idea 2: Design an algorithm that always give a correct answer, runs mostly in polynomial-time (but sometimes runs in exponential time)

Approximation Algorithm

- Target: runs in polynomial time
- Give-ups: may not find optimal solution …
  - Yet, we want to show that the solution we find is "close" to optimal
- E.g., in a maximization problem, we may have an algorithm that always returns a solution at least half the optimal
- How can we do that ??
  - (when we don't even know what optimal is ??)

Example : Min Vertex Cover

- Given a graph $G = (V,E)$, we want to select the minimum # of vertices such that each edge has at least one vertex selected
- Real-life example:
  - edge: road
  - vertex : road junction
  - selected vertex: guard

- This problem is NP-hard

Example : Min Vertex Cover

- Let us consider the following algorithm:

1. C = an empty set
2. while (there is edge in G) {

    Pick an edge, say (u,v) ;

    Put u and v into C ;

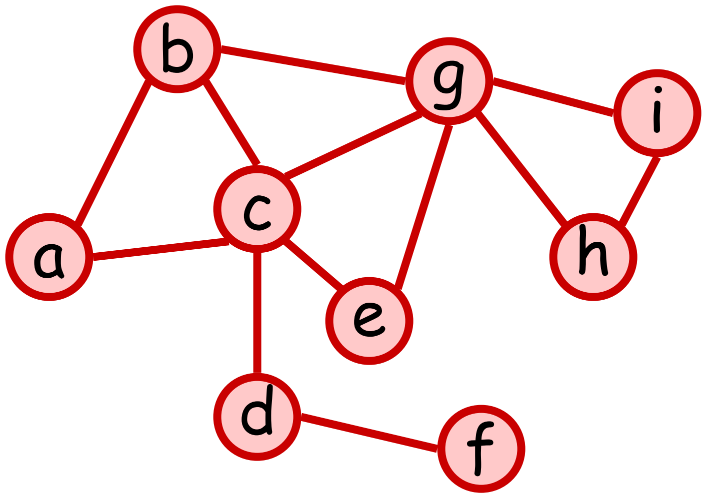
    Remove u, v, and all edges adjacent to u or v ;

   }

3. return C

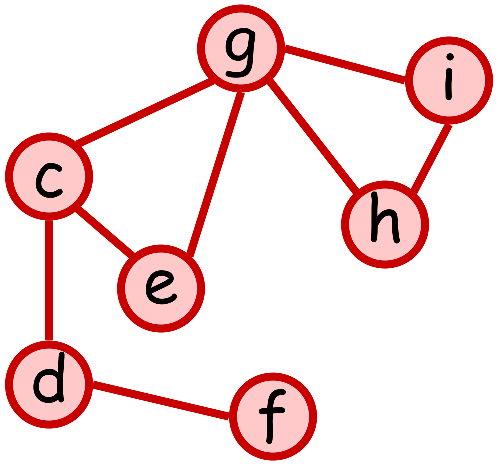
Example Run

original G



Example Run

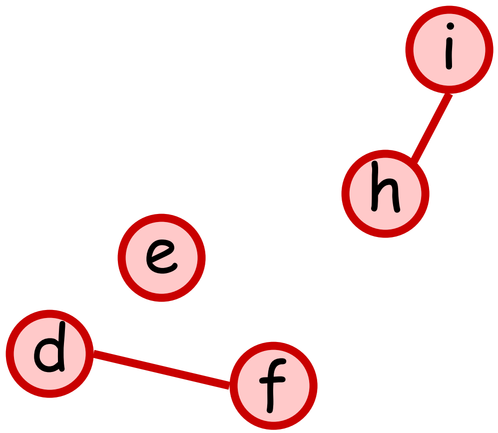
Picking (a,b)



$C = \{ a, b \}$

Example Run

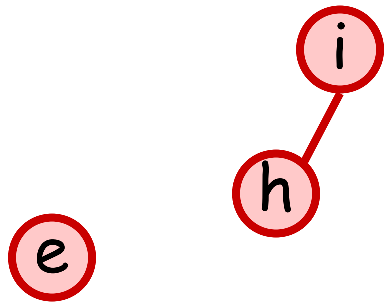
Picking (c,g)



C = { a, b, c, g }

Example Run

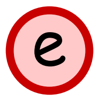
Picking (d,f)



C = { a, b, c, g, d, f }

Example Run

Picking (h,i)



C = { a, b, c, g, d, f, h, i }

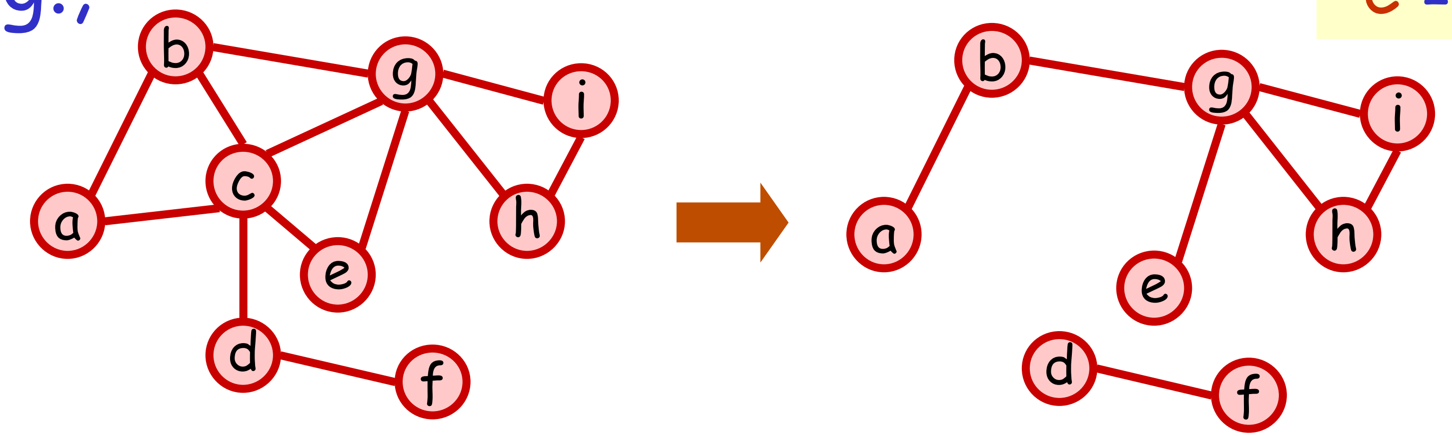
Example : Min Vertex Cover

- What is so special about *C* ?
  - Vertices in *C* must cover all edges !!
  - But … it may not be the smallest one
- How far is it from the optimal ?
  - At most 2 times (why??)
  - Because each edge can only be covered by its endpoints ➔ in each iteration, one of the selected vertex **must be** in the optimal vertex cover

Example : Min Vertex Cover

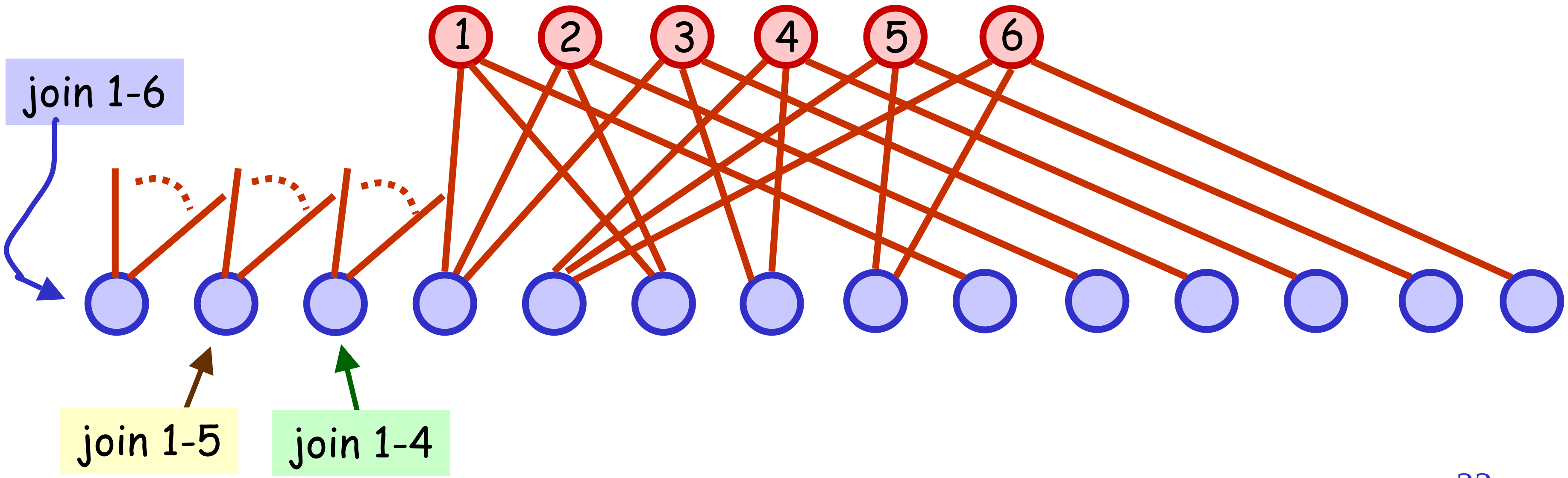
- Another algorithm, perhaps a more natural one, is to select the vertex that covers most edges in each iteration
  - After the selection, we remove the vertex, and all its adjacent edges
- E.g.,

C = { c }



- Unfortunately, when the input graph has n vertices, this new algorithm can only guarantee a cover at most O(log n) times the optimal (instead of at most 2 times before)
- A worst-case scenario looks like :

Optimal : 6 nodes (red) New algo : 13 nodes (blue)



Example : Max-Cut

- Given a graph $G = (V,E)$, we want to partition V into disjoint sets (V_1,V_2) such that #edges in-between them (I.e., with exactly one end-point in each set) is maximized
  - (V_1,V_2) is usually called a cut
  - target: find a cut with maximum #edges

- This problem is NP-hard

Example : Max-Cut

Fact: If the graph has m edges, the maximum #edges in any cut is m

- Thus, if we can find a cut which has at least m/2 edges, this will be at least half of the optimal
- How to find this cut ?

- Let us consider the following algorithm:

1. $V_1 = V_2 =$ empty set ;
2. Label the vertices by $x_1, x_2, \ldots, x_n$
3. For (k = 1 to n) {

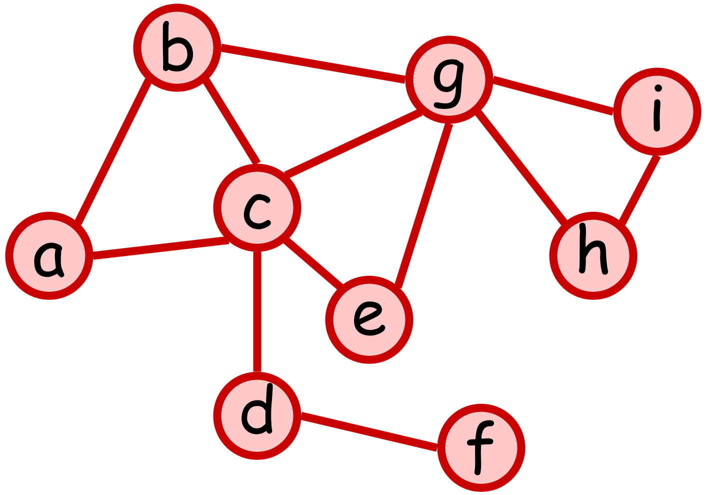
   ```
   /* Fix location of xk */
   ```

   Fix x_k to the set such that more in-between edges (with those already fixed vertices $x_1, x_2, \ldots, x_{k-1}$) are obtained ;

   }
4. return the cut (V_1,V_2) ;

Example Run

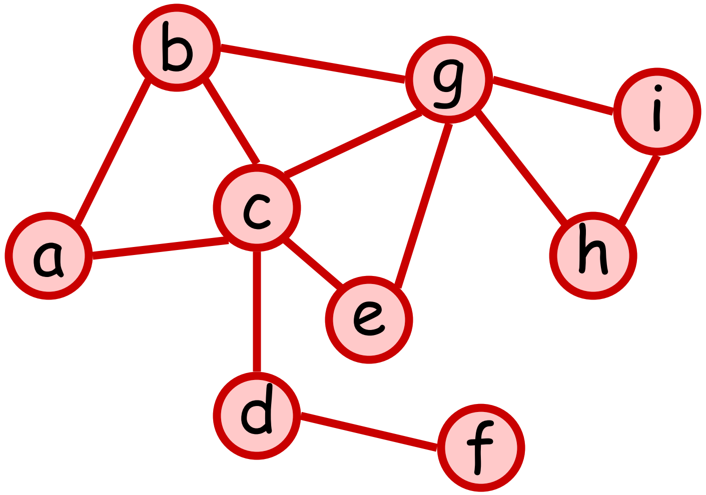
original G



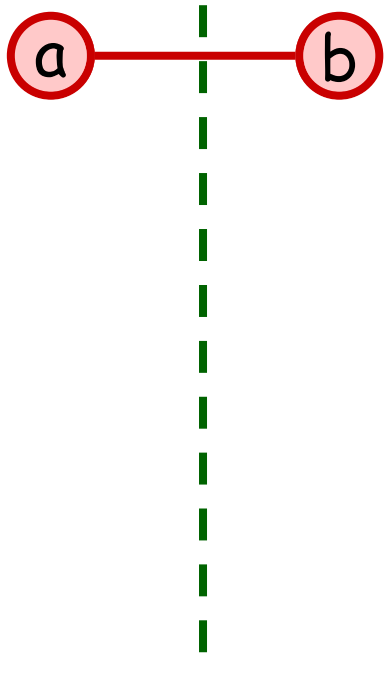
Fix vertex a

Example Run

original *G*

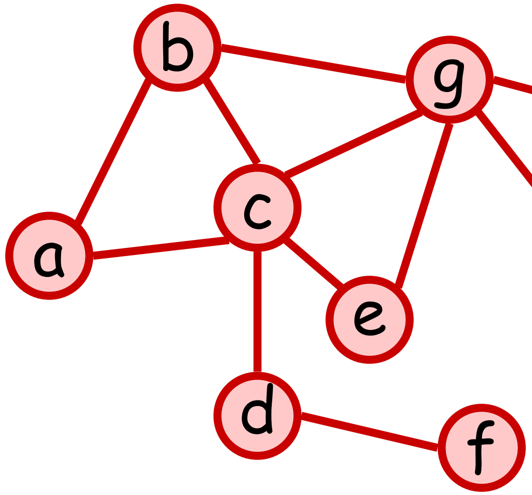


Fix vertex b

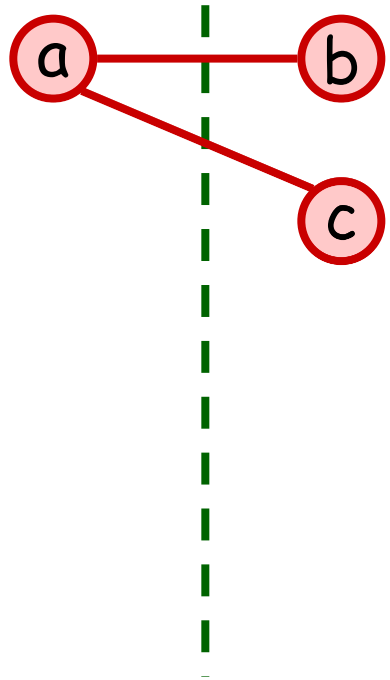


Example Run

original G



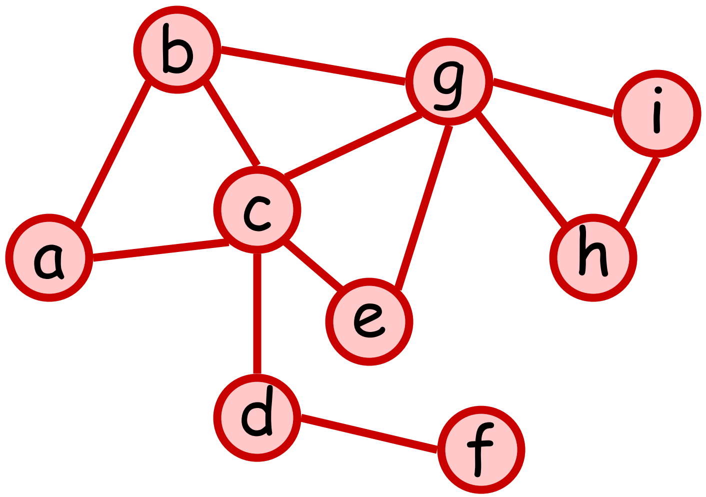
Fix vertex c



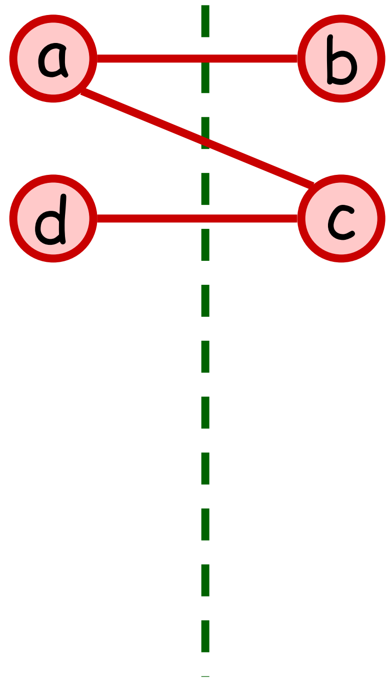
vertex c can be added to either side

Example Run

original G

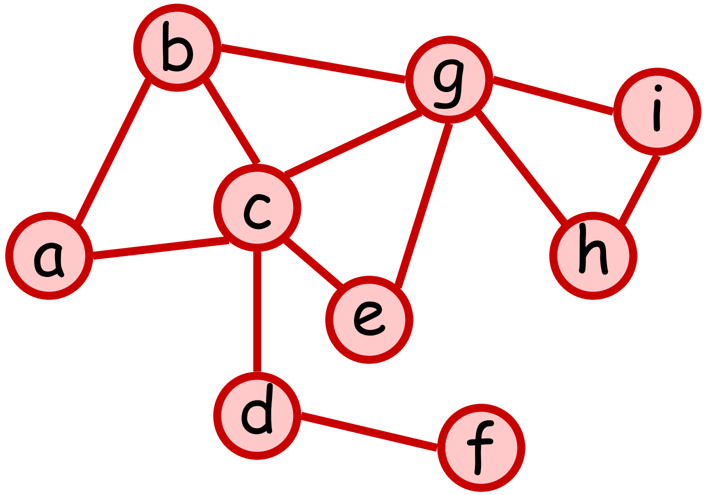


Fix vertex d

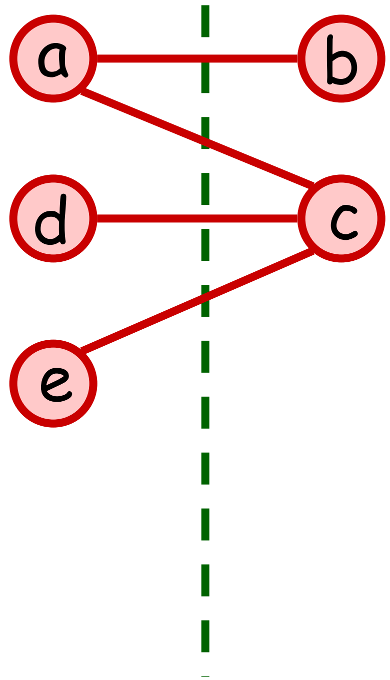


Example Run

original *G*

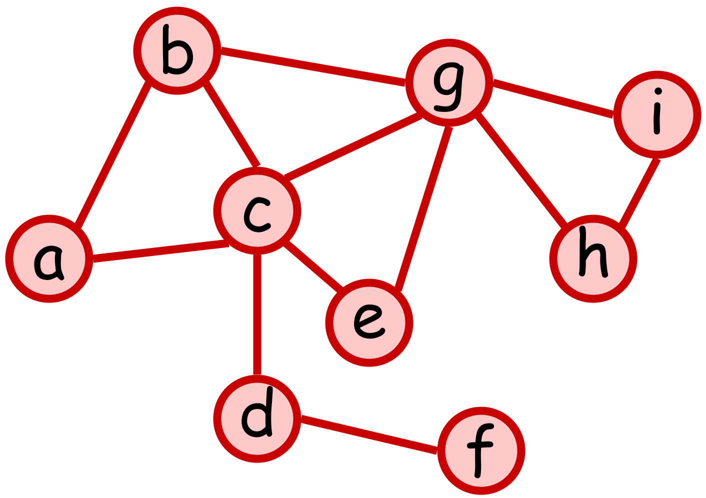


Fix vertex *e*

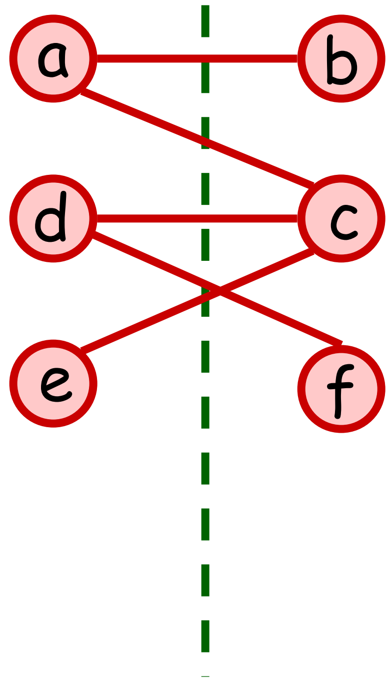


Example Run

original G

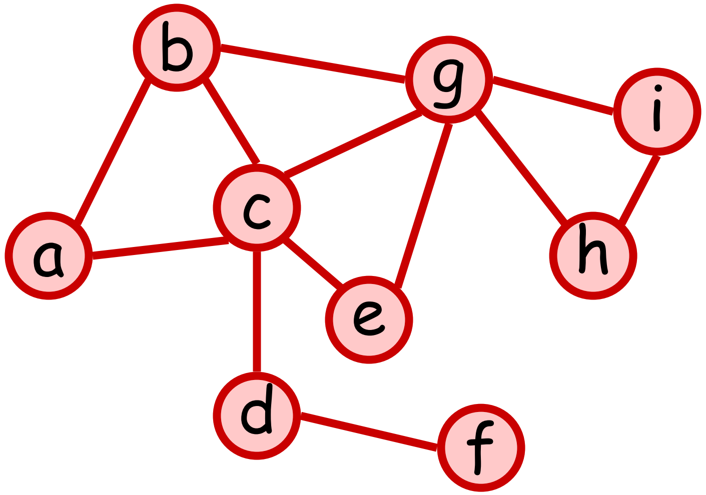


Fix vertex f

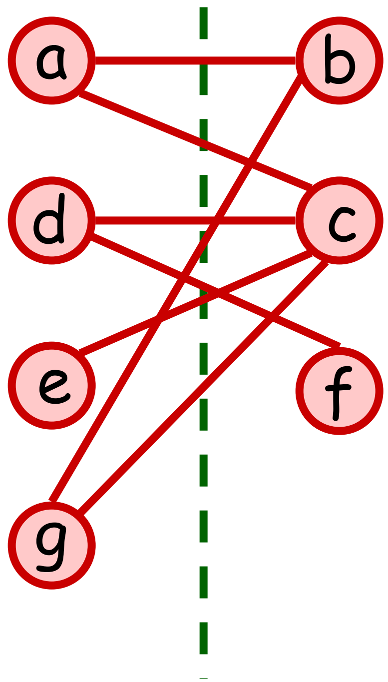


Example Run

original G

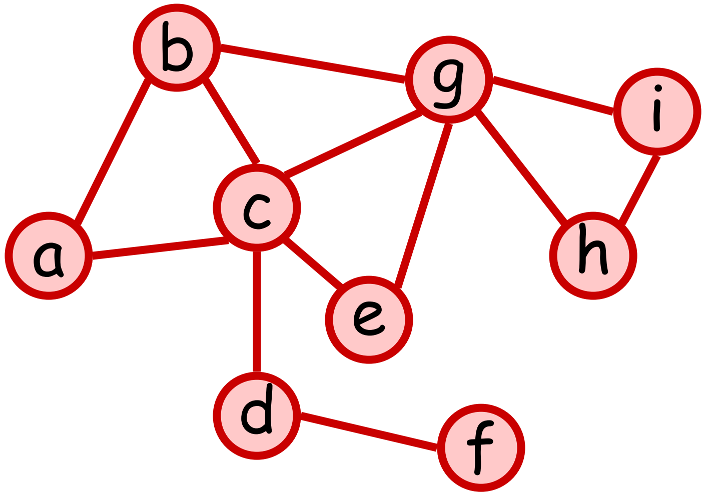


Fix vertex g

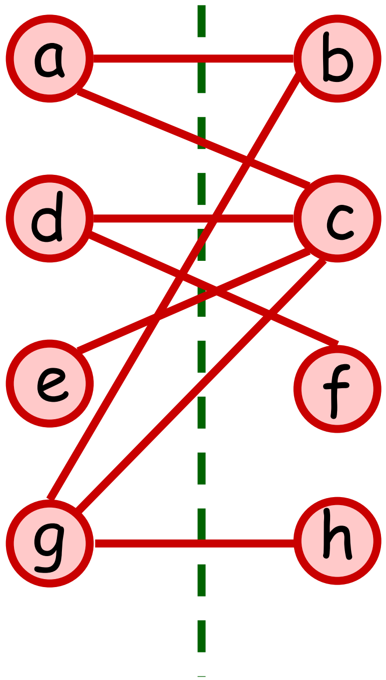


Example Run

original G

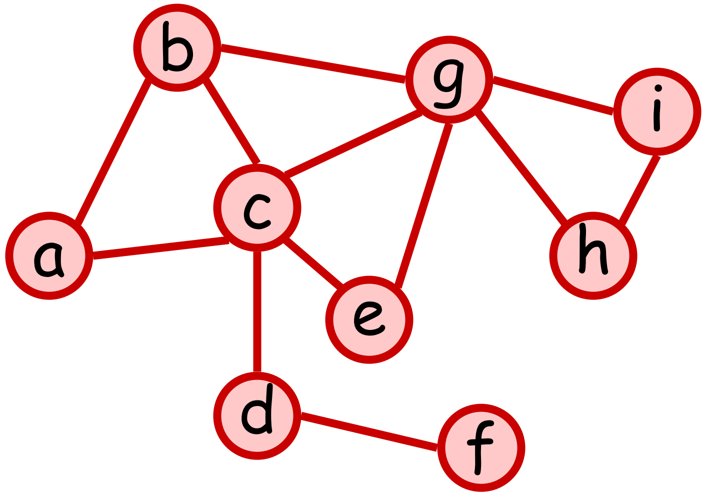


Fix vertex h

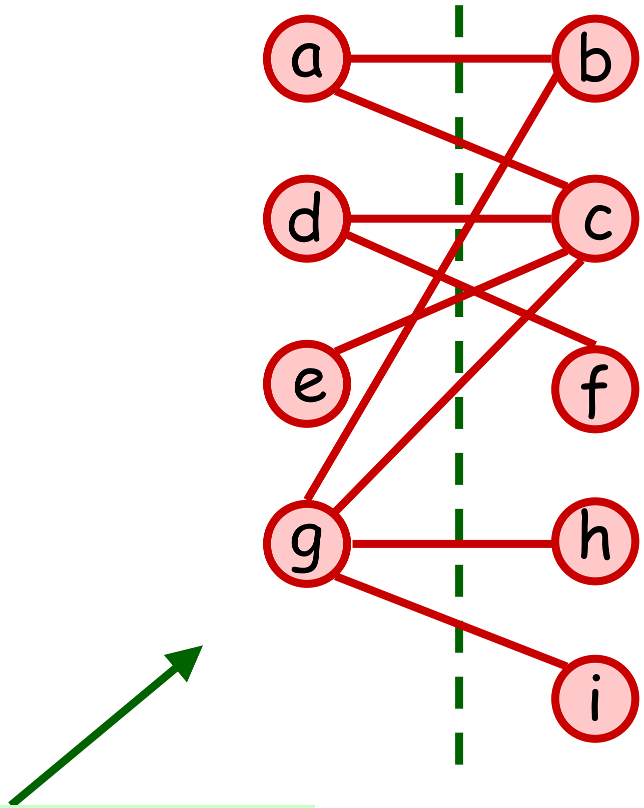


Example Run

original G



Fix vertex i



#in-between edges = 9

Example : Max-Cut

- How far is our cut from the optimal ?
  - At most 2 times (why??)
  - When a vertex v is fixed, we will add some edges into the cut, and discard some edges (u,v) if u is placed in the same set as v
  - But when each vertex is fixed :

    #edges added $\geq$ #edges discarded

    ➔ total #edges added $\geq$ m/2