Hopcroft Karp

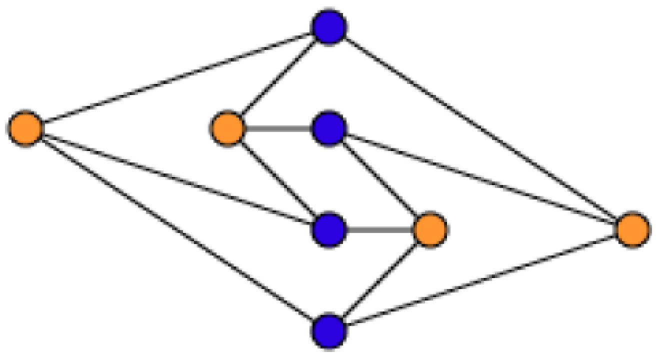
CONTENTS

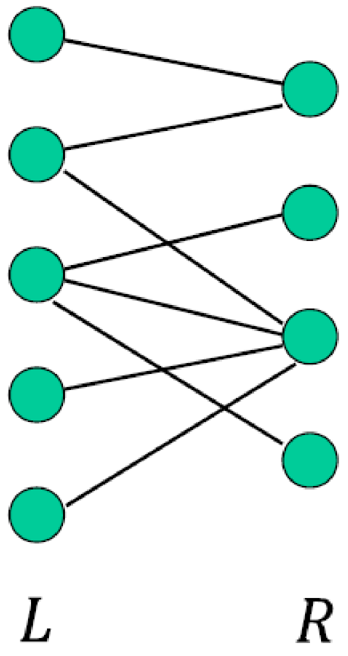
Ⅰ. Hopcroft-Karp

Ⅰ. Hopcroft-Karp

1. Bipartite Matching

- 한글로 번역하면 **이분 매칭**
- **이분 그래프(=Bipartite Graph)에서의 최대 매칭**을 찾는 문제이다.
- 그렇다면 이분 그래프가 무엇일까??
- 그림으로 그리면 아래와 같다.

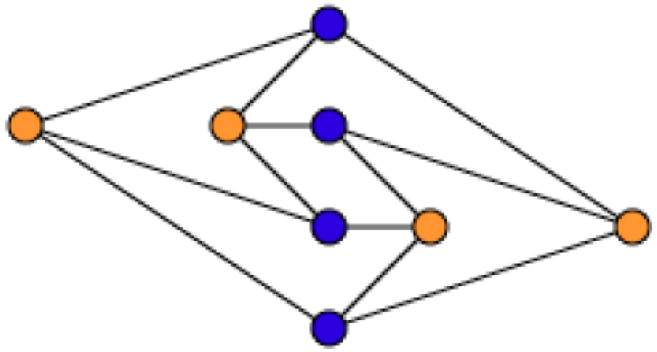


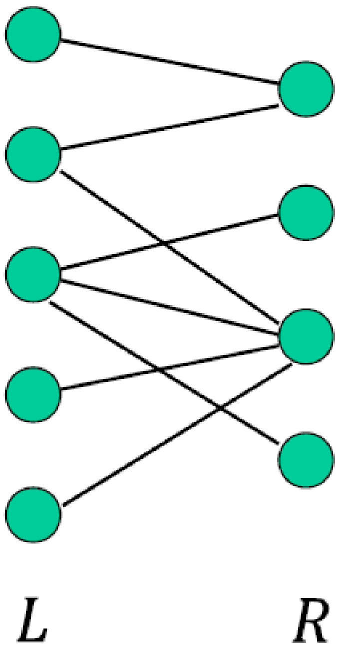


Ⅰ. Hopcroft-Karp

1. Bipartite Matching

- 정확한 **정의**는 아래와 같다.
- 이분 그래프 $G(V, E)$는 **집합 V**가 2개의 **서로 교집합 없는 집합 L과 R**로 나누어 질 수 있으며, $(u, v) \in E$는 $u \in L$이고 $v \in R$ 또는 $u \in R$이고 $v \in L$을 나타내는 그래프이다.

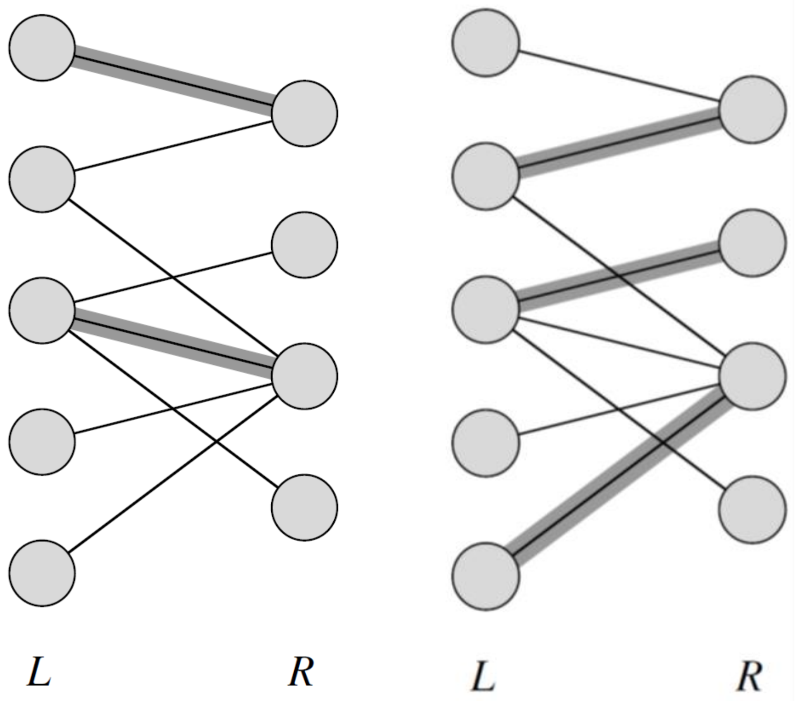




Ⅰ. Hopcroft-Karp

1. Bipartite Matching

- 그래프에서 **매칭**이란, 다음을 만족하는 **부분집합 $\boldsymbol{M}$**을 의미한다.
- 그래프 $G(V,E)$에 대하여, **$\boldsymbol{M} \subset \boldsymbol{E}$**이고 **모든 정점 $\boldsymbol{v} \in \boldsymbol{V}$에 대하여 정점 $\boldsymbol{v}$를 수반한 간선은 집합 $\boldsymbol{M}$에 최대 1개 존재**한다.
- 즉, 이분 매칭은 오른쪽 그림과 같은 상황을 의미한다.
- 오른쪽 예시의 경우 2번째 그림이 최대 매칭을 보여주고 있다. ($|M|$의 최댓값 = 3)



Ⅰ. Hopcroft-Karp

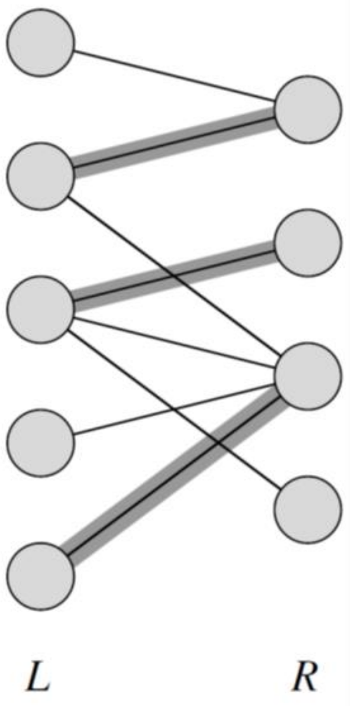
1. Bipartite Matching

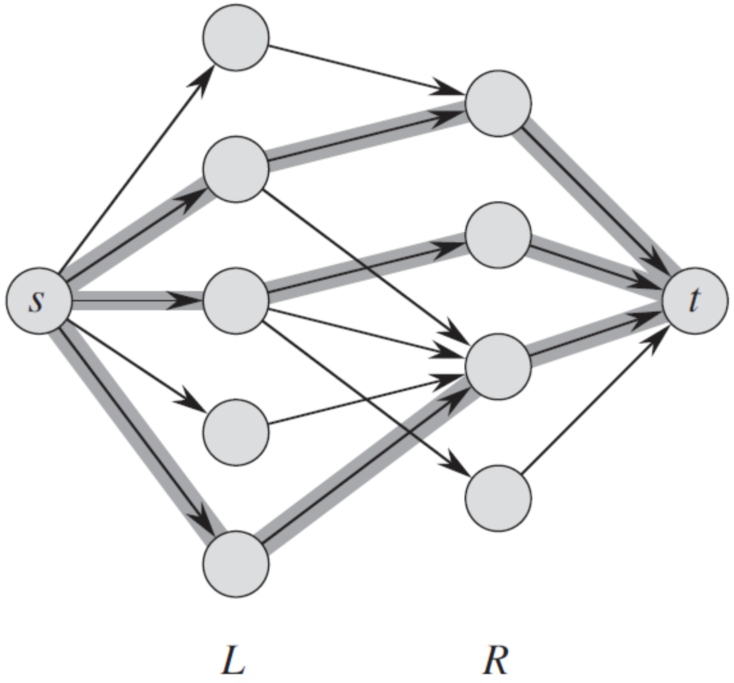
- 그렇다면 Maximum Bipartite Matching을 **어떻게 구할 수 있을까??**
- 지난주에 다룬 **Network Flow를 사용**하면 된다.
- 즉, **적당한 Flow Network를 구성**하면 될 것이다.
- 방법은 간단한데 **Source에서 L로 가는 간선**과, **L에서 R로 가는 간선**, 마지막으로 **R에서 Sink로의 간선**을 만들면 된다. 단, 이때 **각 간선의 capacity는 모두 1**이다.

Ⅰ. Hopcroft-Karp

1. Bipartite Matching

- 방법은 간단한데 **Source에서 *L*로 가는 간선**과, ***L*에서 *R*로 가는 간선**, 마지막으로 ***R*에서 Sink로의 간선**을 만들면 된다. 단, 이때 **각 간선의 capacity는 모두 1**이다.
- 아래는 이를 통해 만든 Flow Network의 예시이다.





Ⅰ. Hopcroft-Karp

1. Bipartite Matching

- 아래와 같이 Flow Network를 만든 뒤에는, Dinic과 같은 Maximum Flow를 찾아주는 알고리즘을 사용하여 문제를 해결할 수도 있지만..
- 그래프를 만드는데 시간이 많이 걸리고 코드도 길기 때문에 **Maximum Bipartite Matching만을 구하는데 특화된 알고리즘**이 존재하고, PS대회에 출제 가능한 가장 빠른 알고리즘이 **Hopcroft-Karp** 알고리즘이다.

Ⅰ. Hopcroft-Karp

2. $O(|V||E|)$ Bipartite Matching

- 바로 Hopcroft-Karp 알고리즘에 대하여 알아보기 보다는 가장 기본적이고 쉬운 버전의 **$O(|V||E|)$짜리 알고리즘**에 대하여 알아보도록 하자.
- 이분 매칭 역시 Network Flow의 일종이므로 **일단 Flow를 흘려 보내고, 수정하는 식의 형태**를 지닌다.
- 의사코드를 알아보도록 하자.

Ⅰ. Hopcroft-Karp

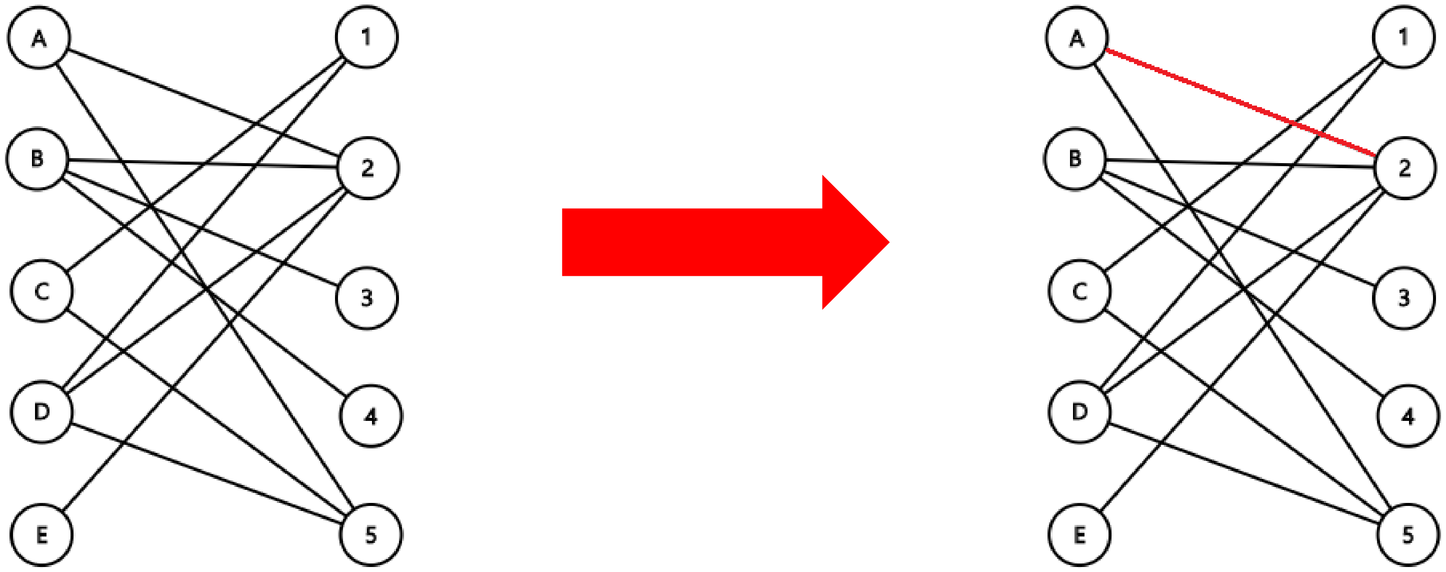
2. $O(|V||E|)$ Bipartite Matching

- 모든 정점은 최대 1번만 매칭 될 수 있으므로, **집합 L에 속한 정점들을 순서대로 조사해보면서 매칭이 가능한지 확인**하면 된다.
- 즉, **각 정점에 대하여 Augmenting path가 존재하는지 여부를 확인**하여 존재한다면 flow를 흘리고, **1만큼 전체 매칭 수가 늘어난다.**

Ⅰ. Hopcroft-Karp

2. $O(|V||E|)$ Bipartite Matching

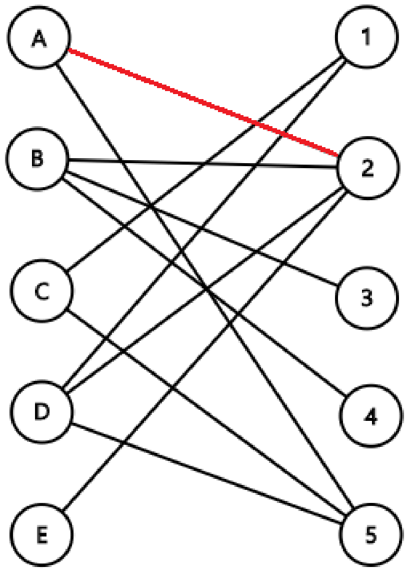
- 즉, **각 정점에 대하여 Augmenting path가 존재하는지 여부를 확인**하여 존재한다면 flow를 흘리고, **1만큼 전체 매칭 수가 늘어난다.**
- 맨 처음 그래프는 아래와 같고 A의 경우 2와 매칭이 가능하다.

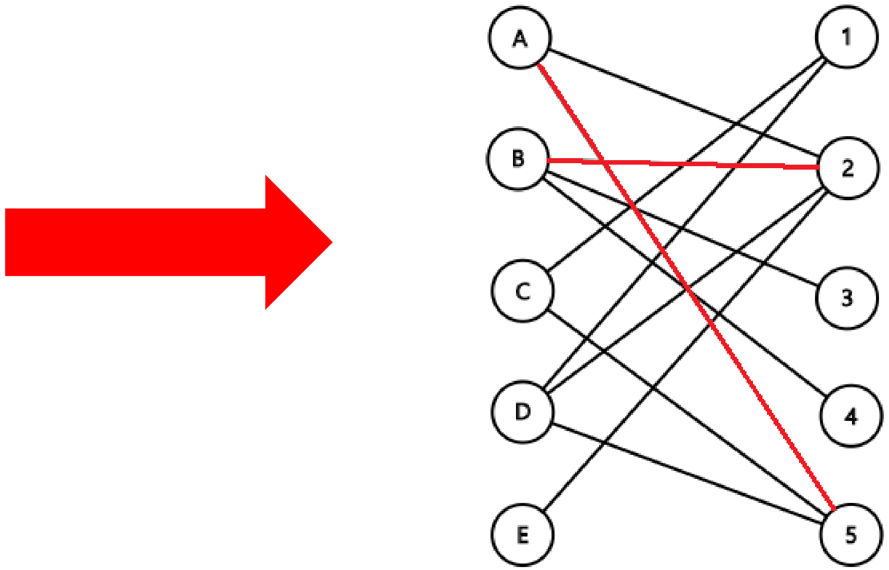


Ⅰ. Hopcroft-Karp

2. $O(|V||E|)$ Bipartite Matching

- 즉, **각 정점에 대하여 Augmenting path가 존재하는지 여부를 확인**하여 존재한다면 flow를 흘리고, **1만큼 전체 매칭 수가 늘어난다.**
- B의 경우 B -> 2 -> A -> 5라는 Augmenting path가 존재하므로 다음과 같이 된다.

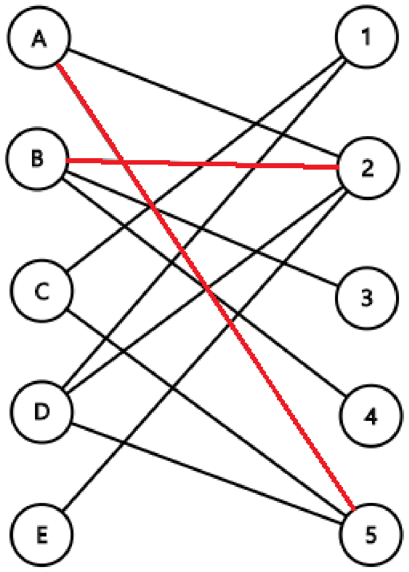


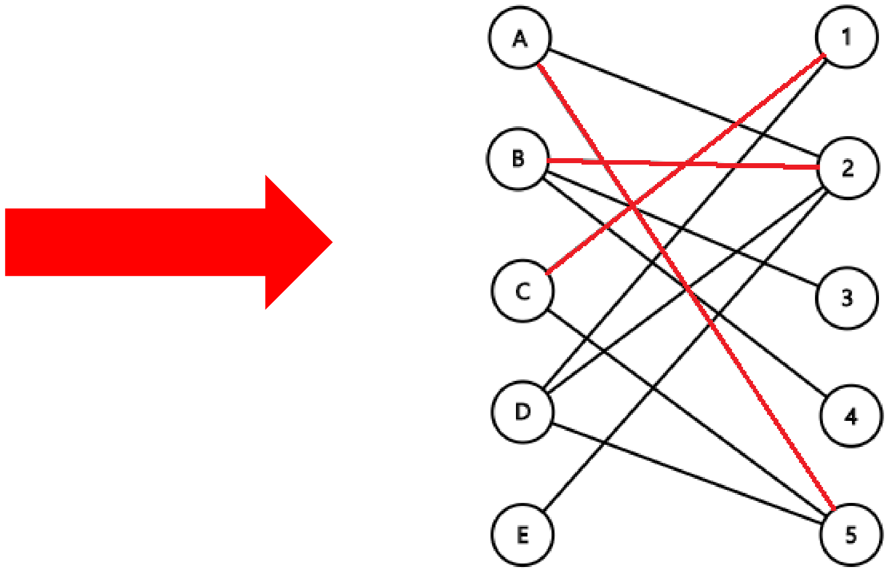


Ⅰ. Hopcroft-Karp

2. $O(|V||E|)$ Bipartite Matching

- 즉, **각 정점에 대하여 Augmenting path가 존재하는지 여부를 확인**하여 존재한다면 flow를 흘리고, **1만큼 전체 매칭 수가 늘어난다.**
- C의 경우 1과 매칭이 가능하다.

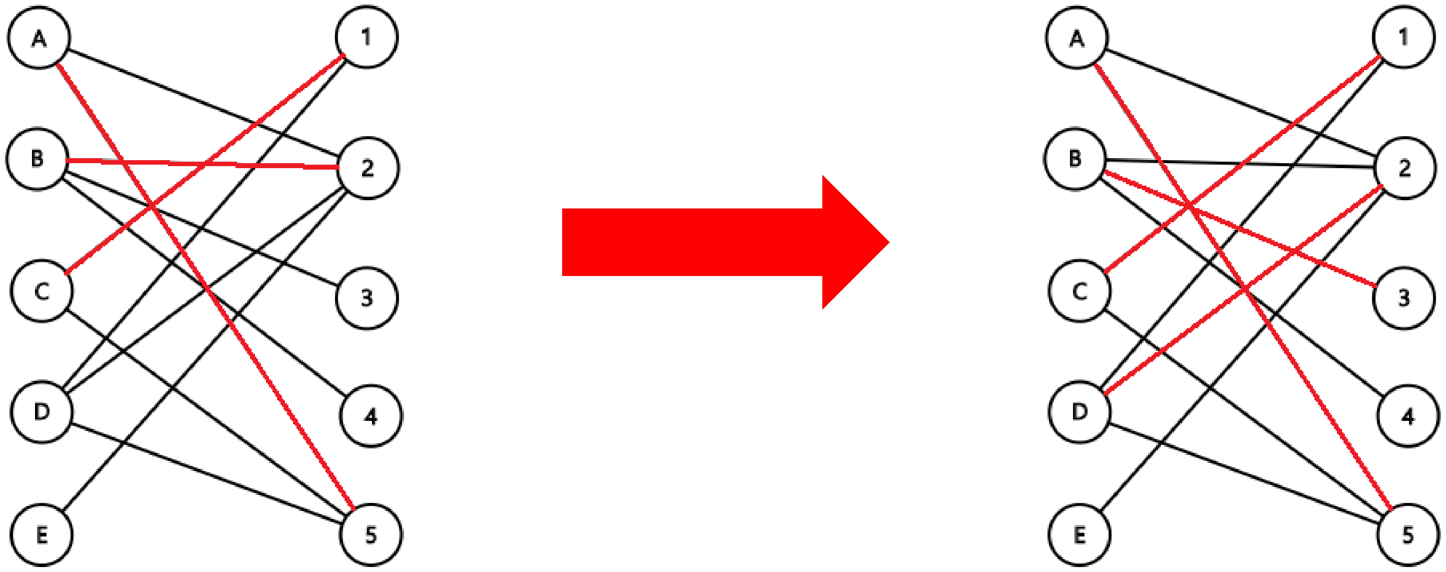




Ⅰ. Hopcroft-Karp

2. $O(|V||E|)$ Bipartite Matching

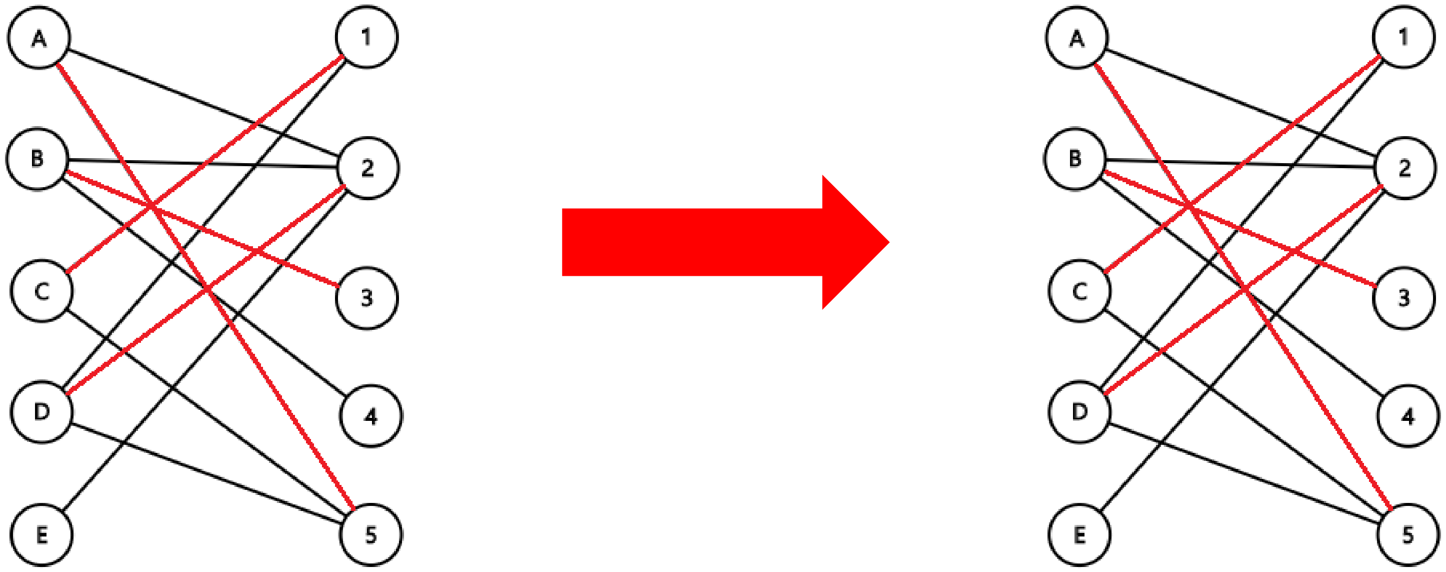
- 즉, **각 정점에 대하여 Augmenting path가 존재하는지 여부를 확인**하여 존재한다면 flow를 흘리고, **1만큼 전체 매칭 수가 늘어난다.**
- D의 경우 D -> 2 -> B -> 3이란 Augmenting path가 존재하므로 다음과 같이 된다.



Ⅰ. Hopcroft-Karp

2. $O(|V||E|)$ Bipartite Matching

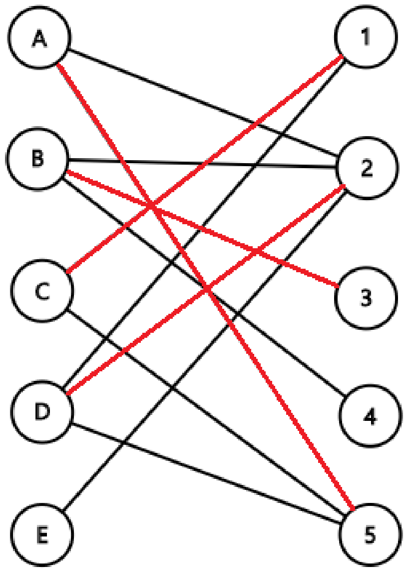
- 즉, **각 정점에 대하여 Augmenting path가 존재하는지 여부를 확인**하여 존재한다면 flow를 흘리고, **1만큼 전체 매칭 수가 늘어난다.**
- E의 경우 더 이상 Augmenting path가 존재하지 않으므로 그대로이다.



Ⅰ. Hopcroft-Karp

2. $O(|V||E|)$ Bipartite Matching

- 즉, **각 정점에 대하여 Augmenting path가 존재하는지 여부를 확인**하여 존재한다면 flow를 흘리고, **1만큼 전체 매칭 수가 늘어난다.**
- 따라서 최종 Maximum Bipartite Matching은 4가 된다.



Ⅰ. Hopcroft-Karp

2. $O(|V||E|)$ Bipartite Matching

- 그렇다면 이 알고리즘의 시간 복잡도는 어떻게 될까??
- 계산이 복잡하지만, 간단하게 결론만 말하면 Network Flow를 구하는 알고리즘 중 **Edmonds-Karp 알고리즘과 동일하게 동작**하기 때문에 $\mathbf{min}(\boldsymbol{O}(|\boldsymbol{V}||\boldsymbol{E}|^2), \boldsymbol{O}(|\boldsymbol{E}|\boldsymbol{f}))$**의 시간 복잡도**를 갖는다.
- 이때, f가 해당 Flow Network의 Maximum Flow를 의미하므로 **이분 매칭에서는** $\boldsymbol{f} = |\boldsymbol{V}|$이다. 따라서 $\min(O(|V||E|^2), O(|V||E|)) =$ $\boldsymbol{O}(|\boldsymbol{V}||\boldsymbol{E}|)$가 된다.

Ⅰ. Hopcroft-Karp

2. $O(|V||E|)$ Bipartite Matching

- 잠시 구현을 보면 오른쪽과 같다.
- **lst에 $L \rightarrow R$ 방향의 간선만** 넣도록 한다.
- 코드에서는 **G1과 G2가 각각 L과 R**을 나타낸다.
- 최종적으로 **match가 최종 flow 값**을 가지고 있으며, **G1에 G1의 각 정점이 G2의 어떤 정점에 매칭되어 있는지**가 적혀 있게 된다.
- 나중에 **Hopcroft-Karp에서 이와 유사한 코드를 보게 될 것**이다.

```cpp
#include<bits/stdc++.h>

using namespace std;

int n, m;
vector<int> lst[102];
int G1[102], G2[102];
bool visit[102];
//size of lst, G1 and visit is MAXN
//size of G2 is MAXM
//vertex number should be 1~n in G1 and 1~m in G2
// only add edges which is from G1 to G2

bool dfs(int v1)
{
    visit[v1]=true;
    for(int i=0 ; i<lst[v1].size() ; i++){
        int v2=lst[v1][i];
        if(G2[v2]==-1 || (!visit[G2[v2]] && dfs(G2[v2]))){
            G1[v1]=v2;
            G2[v2]=v1;
            return true;
        }
    }
    return false;
}

int main()
{
    int match=0;
    memset(G1,-1,sizeof(G1));
    memset(G2,-1,sizeof(G2));
    for(int i=1 ; i<=n ; i++){
        if(G1[i]==-1){
            memset(visit,false,sizeof(visit));
            if(dfs(i)) match++;
        }
    }
    return 0;
}
```

Ⅰ. Hopcroft-Karp

3. 개념

- 바로 Hopcroft-Karp를 설명하면 될 텐데, 굳이 $O(|V||E|)$짜리 이분 매칭 알고리즘을 소개한 이유는 바로, **$O(|V||E|)$짜리 이분 매칭 알고리즘에 Dinic의 핵심 아이디어인 Level Graph를 도입하면 Hopcroft-Karp**가 되기 때문이다.

Ⅰ. Hopcroft-Karp

3. 개념

- 좀 더 자세히 설명하면 Dinic과 같이 Level Graph를 만드는데, 단 Source를 넣는 Dinic과 달리 **현재 시점에서 매칭이 안된 모든 정점에 대하여 Level을 0으로 계산하고 BFS를 통해 나머지 정점에 대하여 계산**을 한다.
- 이후 Dinic과 마찬가지로 현재의 Level Graph를 기준으로 **level 차이가 최대 1만큼 나는 정점으로만 이동**을 하는 식으로 flow를 최대한 흘려본다. 단, 이때 **앞서 설명한 $O(|V||E|)$방식의 이분 매칭 알고리즘을 사용**한다.
- 이 **과정을 반복하여 더 이상 흘릴 수 있는 flow가 없을 때까지 진행**하면 된다.

Ⅰ. Hopcroft-Karp

3. 개념

- 이렇게 하면 시간 복잡도가 어떻게 줄어들까??
- Dinic의 아이디어를 가져왔으니 $O(|V||E|)$보단 빠를 것을 기대할 것이다.
- 증명은 잘 모르지만, 최종 결론만 말하자면 Hopcroft-Karp 알고리즘의 시간 복잡도는 $\boldsymbol{O(|E|\sqrt{|V|})}$가 된다.
- 사실상 PS 대회에 출제 가능한 가장 빠른 Maximum Bipartite Matching 알고리즘이며, **해당 알고리즘으로 시간초과가 나면 다른 알고리즘을 사용하는 문제라고 봐도 무방**하다.