그래프 알고리즘

그래프 용어

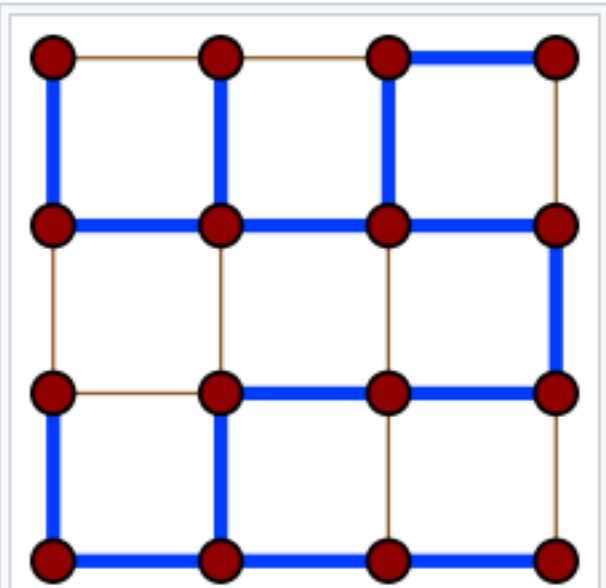
- 그래프 G=(V, E)는 정점 V와 간선 E의 집합
- 경로(path): 간선에 의해 연속적으로 연결된 정점의 리스트 혹은 간선의 리스트.
- 단순경로(simple path): 동일 정점이 중복되어 나타나지 않는 경로.
- 연결그래프: 모든 정점간에 경로가 존재하는 그래프
- 연결성분(connected component): 그래프 내의 연결된 부분그래프

그래프 용어

- 사이클(cycle): 첫째와 마지막 정점이 동일한 경로. 회로(circuit)라고도 함.
- 나무(tree): 사이클이 없는 연결된 그래프.
- 숲(forest): 여러 나무들로 이루어진 그래프.
- 신장나무(spanning tree): 그래프의 모든 정점을 포함하고 있는 나무.
- 완전 그래프(complete graph): 있을 수 있는 모든 간선을 갖는 그래프.

그래프 용어

- 가중 그래프(weighted graph): 간선에 가중치가 주어진 그래프.
- 방향 그래프(directed graph): 간선에 방향이 부여된 그래프.
- 네트워크(network): 가중 방향 그래프



A spanning tree (blue heavy edges) of a grid graph

https://en.wikipedia.org/wiki/Spanning_tree

그래프 표현

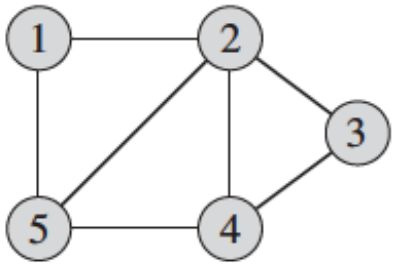
- 그래프 G=(V,E)를 표현하기 뒤한 두가지 표준 방법

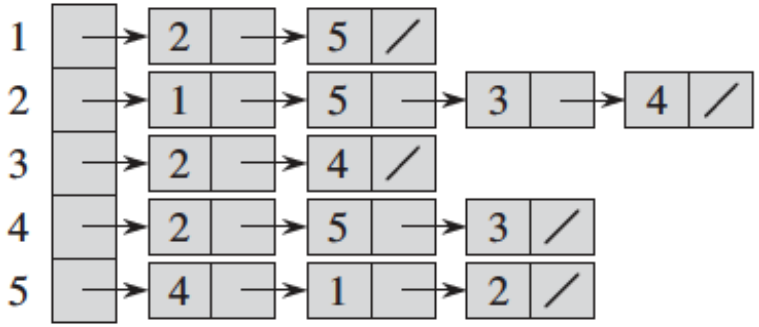
  - 인접 리스트 (adjacency-list) 표현법– 작은 밀도 그래프 ($|E| << |V|^2$)에 대해 효율적인 방법을 제공하여 자주 사용 됨.

    $\Theta(V+E)$ 의 메모리 사용.
  - 인접 행렬 (adjacency-matrix) 표현법 – 높은 밀도 그래프 ($|E| \approx |V|^2$)나 주어진 두 정점을 연결하는 간선이 있는지 여부를 빠르게 확인 할 필요가 있을 때 사용 됨. $\Theta(V^2)$ 메모리 사용 함. Symmetric하므로 반만 사용하면 됨.

그래프 표현

- 무 방향 그래프



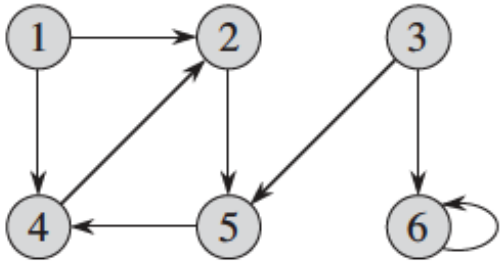


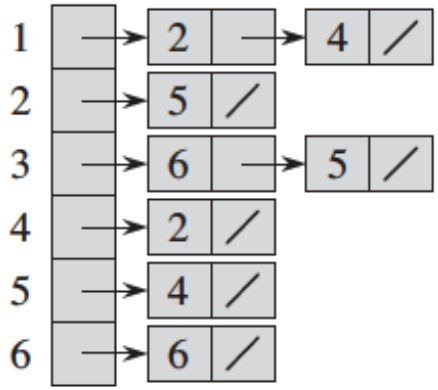
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

- |V|개 리스트의 배열 Adj 로 구성되어 있고 여기서 각 리스트는 V에 들어 있는 정점 하나에 대한 것이다. u∈ V 에 대해 인접 리스트 Adj[u]는 간선 (u,v) ∈ E가 존재하는 모든 정점 v를 포함한다. 즉, Adj[u]는 그래프 G에서 정점 u에 인접해 있는 모든 정점으로 구성된다.
- 인접 리스트들 길이의 총합은 2|E|이다.

그래프 표현

- 방향 그래프





	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

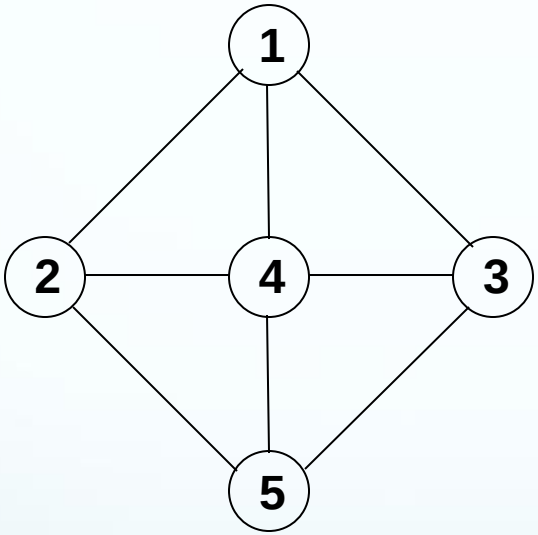
- 정점에 임의의 순서대로 1, 2, ..., |V|의 순서가 매겨져 있다고 가정함. G의 인접행렬 표현접은 다음을 만족하는 |V|x|V| 의 행렬 $A=(a_{ij})$가 된다.

$$a_{ij} = \begin{cases} 1 & (i,j) \in E \text{ 일때} \\ 0 & \text{그 이외의 경우} \end{cases}$$

- 인접 리스트들 길이의 총합은 |E|이다.

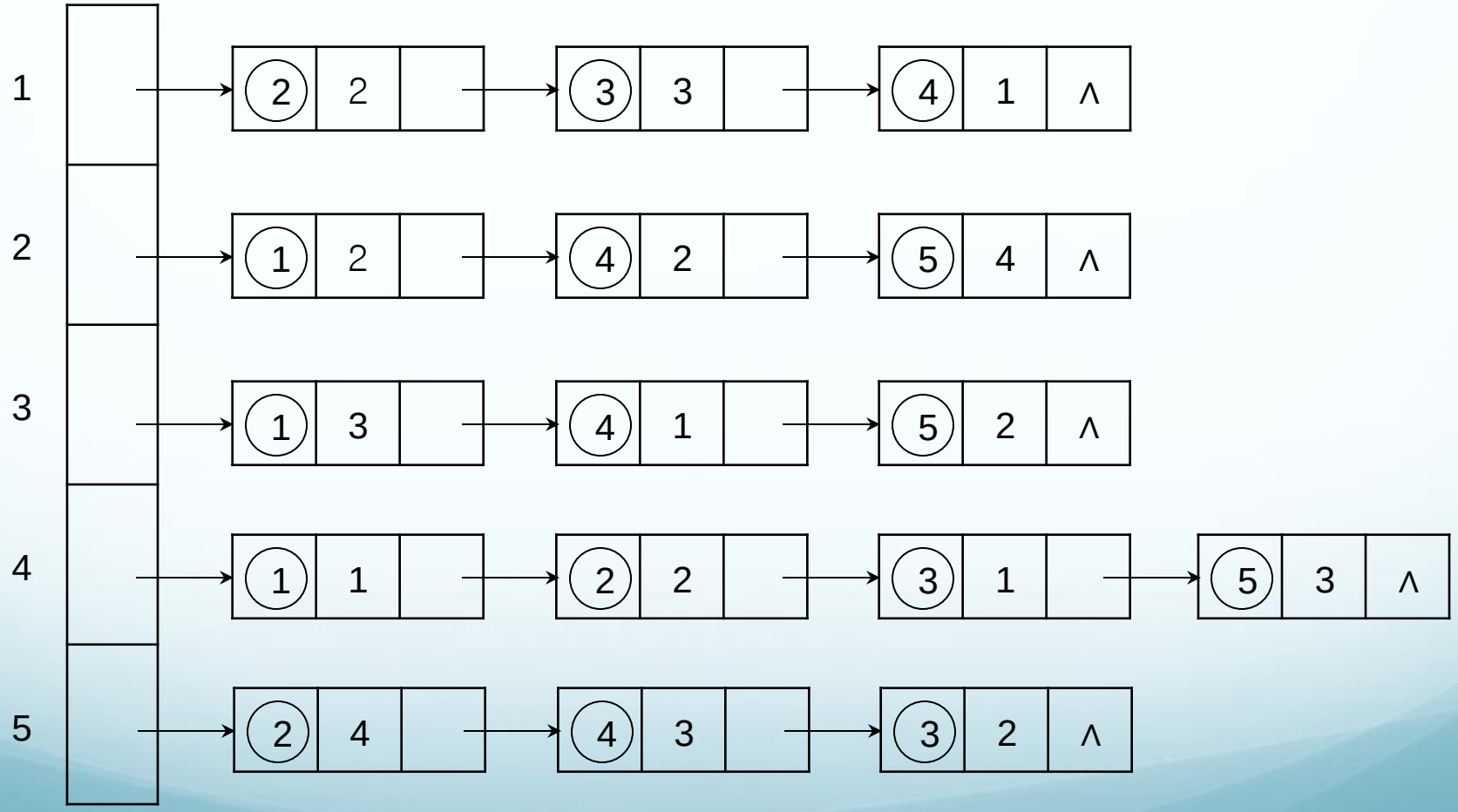
그래프 표현

- 가중치 함수



	1	2	3	4	5
1	0	2	3	1	∞
2	2	0	∞	2	4
3	3	∞	0	1	2
4	1	2	1	0	3
5	∞	4	2	3	0

인접리스트 표현법



```
Admatrix (int n,  int m,  int a[ ][ ])
  입력 : n – 정점의 개수, m – 간선의 개수
  A[ ][ ] – 인접 행렬
  {
1       int u, v, i, j ;
2       for (i = 1;  i <= n;  i++) /* 행렬의 초기화 */
3           for (j = 1;  j <= n;  j++)
4               a[i][j] = 0;
5       for (i = 1;  i <= n;  i++)
6           a[i][i] = 0;
7       for ( i = 1;  i <= m;  i++)  {
8           scanf (“%d  %d”, &u, &v);  /* 간선 (u, v)의 입력 */
9           a[u][v] = 1;
10        a[v][u] = 1;
11      }
    }
```

```
#define  n  50 ;  /*  n = 50 */
struct node {
int vertex ;
float weight ;
struct node *next ;
};
struct node *head[n] ;   /* 포인터 배열 정의 */
```

//head[v] 는 정점 v에 대한 인접 리스트의 처음 노드에 대한 포인터를 저장

```
Adlist (int n, int m, struct node *head[ ])
입력 : n – 노드 수,  m – 간선 수
출력 : head에 의해 지시되는 인접 리스트
{
1       int  u, v, i, j ;
2       float  w ;
3       struct node *t ;
4       for (i = 1;  i <= n;  i++)
5            head[i] = (struct node *) NULL ;
6       for (i = 1;  i <=m;  i++)  {
          /* 간선 (u, v)의 입력 */
7            scanf (" %d  %d  %f", &u,  &v, &w);
          /* 기억 공간 할당 */
8            t = (struct node *) malloc (sizeof (struct  node) ) ;
9            t->vertex = v;  t->weight = w;  t->next = head[u];  head[u] = t;
          /* 기억 공간 할당 */
10           t = (struct node *) malloc(sizeof (sturct node) ) ;
11           t->vertex = u; t->weight = w;  t->next = head[v];  head[v] = t;
12      }
    }
```

//간선 (u,v)가 입력되면 정점 v는 head[u]가 가르키는 u의 인접
//리스트의 처음 노드로 연결되고, 정점 u는 정점 v에 대한
//인접 리스트의 처음 노드로 연결됨.

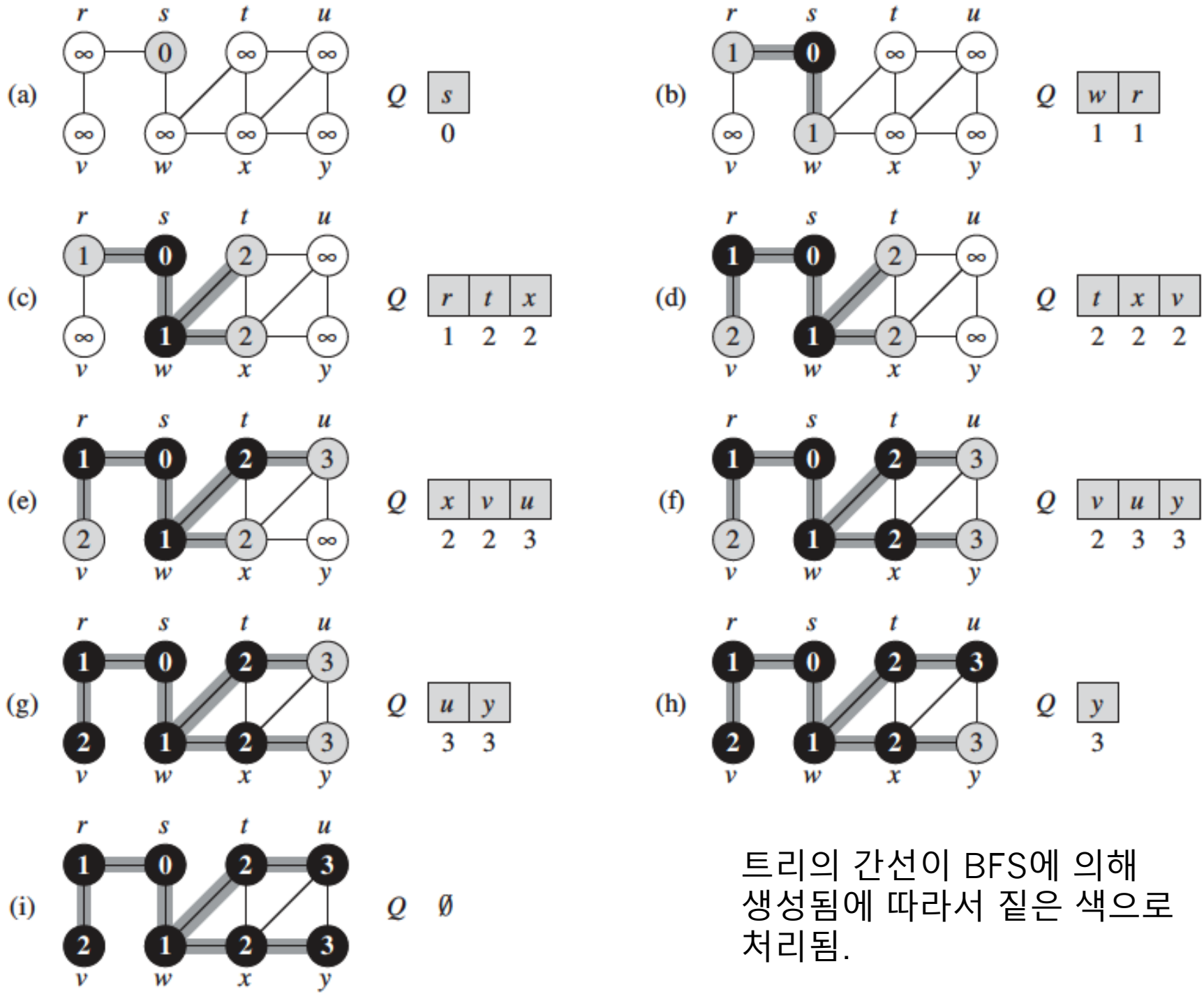
그래프 순회

- 그래프의 모든 정점을 체계적으로 탐색하는 방법
- 너비 우선 탐색(BFS: breadth-first search)
  - 거리 순으로 방문. 가깝게 인접한 정점을 모두 방문한 후 그 다음으로 가깝게 인점한 정점을 방문하는 순으로 진행 함. 거리가 1만큼 떨어진 정점을 모두 방문한 후 거리가 2만큼 떨어진 정점을 모두 방문, etc.
  - 큐(FIFO)로 구현하는 것이 적절함.
- 깊이 우선 탐색(DFS: depth-first search)
  - 정점을 방문할 때 갈 수 있는 데까지 우선 가보다가 더 이상 진행할 수 없으면 왔던 길을 거슬러 올라가면서 아직 가보지 않은 길이 있으면 그 길을 따라 또 갈 수 있는 데까지 가보는 방법.
  - 스택(LIFO)로 구현하는 것이 적절함.

너비우선탐색

BFS: breadth-first search

- 거리 순으로 방문. 가깝게 인접한 정점을 모두 방문한 후 그 다음으로 가깝게 인접한 정점을 방문하는 순으로 진행 함. 거리가 1만큼 떨어진 정점을 모두 방문한 후 거리가 2만큼 떨어진 정점을 모두 방문, etc.
- 큐(FIFO)로 구현하는 것이 적절함.
- 모든 정점은 흰색으로 시작해 회색이 됐다가 검은색이 된다.
  - 흰색 – 발견되지 않은 정점.
  - 검은색 – 인접해 있는 모든 정점이 발견된 것.
- **너비우선 검색은 너비 우선 트리를 만드는데**, 처음 출발점 s만을 루트로 이미 발견된 정점 u의 인접 리스트를 스캔하면서 흰색 정점 v가 발견될 때 마다, 정점 v와 간선 (u,v)가 트리에 더해진다. 이 경우 u를 v의 너비 우선 트리의 직전원소 또는 부모라고 한다.
- **출발정점** s로 **부터 도달할 수 있는 각 정점까지의 최단경로** (, shortest path, 가장 적은 간선의 수)**를 계산**한다.



트리의 간선이 BFS에 의해 생성됨에 따라서 짙은 색으로 처리됨.

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}  //정점 초기화
 2      u.color = WHITE            //정점 색
 3      u.d = ∞                    //거리
 4      u.π = NIL                  //직전원소
 5  s.color = GRAY                 //시작 정점
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅                          //큐
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)             //FIFO로 정점 꺼낸 후
12      for each v ∈ G.Adj[u]      //인접 정점들 검사
13          if v.color == WHITE    //발견되지 않은 노드 있으면
14              v.color = GRAY     //색 바꾸고 거리 늘리고
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)      //큐에 넣음
18      u.color = BLACK //인접 정점 다 발견되었으면 검은색으로
```

너비 우선 검색의 결과는 주어진 정점의 이웃이 12행에서 방문되는 순서에 따라 달라질 수 있다. 단, 이로인해 트리는 달라질 지 모르지만 계산되는 거리 d 는 변하지 않는다.

너비우선탐색

BFS: breadth-first search

- 거리 순으로 방문. 가깝게 인접한 정점을 모두 방문한 후 그 다음으로 가깝게 인접한 정점을 방문하는 순으로 진행 함. 거리가 1만큼 떨어진 정점을 모두 방문한 후 거리가 2만큼 떨어진 정점을 모두 방문, etc.
- 큐(FIFO)로 구현하는 것이 적절함.
- 모든 정점은 흰색으로 시작해 회색이 됐다가 검은색이 된다.
  - 흰색 – 발견되지 않은 정점.
  - 검은색 – 인접해 있는 모든 정점이 발견된 것.
- **너비우선 검색은 너비 우선 트리를 만드는데**, 처음 출발점 s만을 루트로 이미 발견된 정점 u의 인접 리스트를 스캔하면서 흰색 정점 v가 발견될 때 마다, 정점 v와 간선 (u,v)가 트리에 더해진다. 이 경우 u를 v의 너비 우선 트리의 직전원소 또는 부모라고 한다.
- **출발정점** s로 **부터 도달할 수 있는 각 정점까지의 최단경로** (, shortest path, 가장 적은 간선의 수)**를 계산**한다.

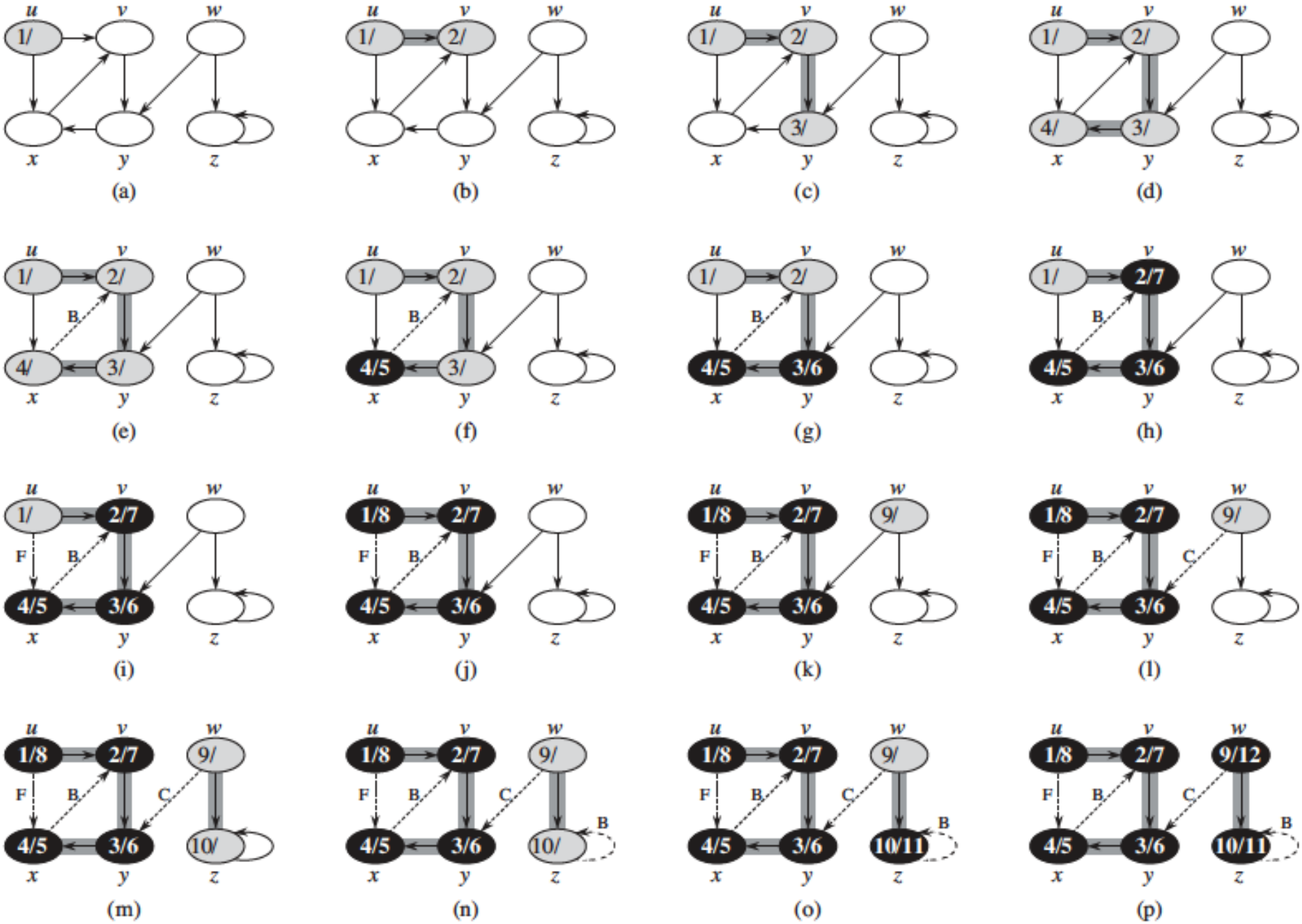
너비우선탐색
BFS: breadth-first search

- BFS가 최단 경로 트리를 계산하였을 때, s에서 v까지의 최단 경로에 있는 정점 출력

```
PRINT-PATH(G, s, v)
1  if v == s
2       print s
3  elseif v.π == NIL
4       print "no path from" s "to" v "exists"
5  else PRINT-PATH(G, s, v.π)
6       print v
```

깊이 우선 탐색
(DFS: depth-first search)

- 정점을 방문할 때 갈 수 있는 데까지 우선 가보다가 더 이상 진행할 수 없으면 왔던 길을 거슬러 올라가면서 아직 가보지 않은 길이 있으면 그 길을 따라 또 갈 수 있는 데까지 가보는 방법.
- 스택(LIFO)로 구현하는 것이 적절함.
- **깊이우선 검색은** 너비우선 검색과 다르게 **깊이 우선 포리스트를 형성한다.** 각 정점이 정확히 하나의 깊이 우선 트리에만 포함되어 트리가 서로 공통인 부분이 없도록 흰색, 회색, 검은색 노드를 사용.
- 각점정에 시간기록을 한다. v.d는 v가 처음 발견되었을때(회색으로 칠할 때)를 기록하고, 두번째 시간기록 v.f는 검색이 v의 인접 리스트 조사를 마쳤을 때(검은색으로 칠할 때)를 기록한다.


u v w
1/
x y z
(a)
1/ 2/
(b)
1/ 2/ 3/
(c)
1/ 2/ 4/ 3/
(d)
B
1/ 2/ 4/ 3/
(e)
1/ 2/ 4/5 3/
(f)
1/ 2/ 4/5 3/6
(g)
1/ 2/7 4/5 3/6
(h)
F B
1/ 2/7 4/5 3/6
(i)
1/8 2/7 4/5 3/6
(j)
1/8 2/7 9/ 4/5 3/6
(k)
C
1/8 2/7 9/ 4/5 3/6
(l)
1/8 2/7 9/ 4/5 3/6 10/
(m)
B
1/8 2/7 9/ 4/5 3/6 10/
(n)
1/8 2/7 9/ 4/5 3/6 10/11
(o)
9/12
1/8 2/7 9/12 4/5 3/6 10/11
(p)

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
 1  time = time + 1          // white vertex u has just been discovered
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]    // explore edge (u, v)
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK          // blacken u; it is finished
 9  time = time + 1
10  u.f = time
```

깊이 우선 검색의 결과는 DFS의 5행의 조사 정점의 순서와 DFS-VISIT 4행의 이웃하는 정점의 방문 순서에 영향을 받을 수 있다. 단 본질적으로 동등한 결과를 가지므로 다른 방문순서는 실제로 문제를 일으키지 않는다.

BFS, DFS의 다른 구현방법

- 인접 리스트 자료구조

```
#define  n  100    /* 정점의 개수  n = 100 */
struct node  {
    int vertex ;
    struct node  *next ;
} ;
struct node  *head[n];
```

DFS

```
int mark, flag[n] ;
  dfg (G){
    //입력 : 인접 리스트 표현의 그래프 G = (V, E),  출력 : 모든 정점에 대한 DFS 방문 결과.  배열 flag에 방문 순서 기록.
1       int v ;
2       mark = 0 ;
3      for (v = 0;  v < n;  v++)  /* flag의 초기화 */
4         flag[v] = 0;
5      for (v=0;  v <n;  v++)
6       if (flag[v] == 0)  dfg_visit (v);
  }
```

DFS

```
dfs_visit (int v)
    //입력 : 정점 v , 출력 : v가 속한 연결성분의 순환적 깊이 우선 탐색
   {
1      struct node *t ;
2      mark++ ;  flag[v] = mark ;
3      t = head ;
4      while (t != NULL)  {
5        if (flag[t->vertex] == 0)  dfs_visit (t->vertex) ;
6        t = t->next ;
7   }
  }
```

BFS

int flag[n], mark ;

bfs (G)
입력 : 인접 리스트 표현의 그래프 G = (V, E)
출력 : 모든 정점에 대한 BFS탐색 결과. 배열 flag에 방문 순서 기록.
{

```
1       int v ;
2       mark = 0 ;
3       init_queue( ) ;
4       for (v = 0;  v < n;  v++)     /* flag의 초기화 */
5           flag[v] = 0 ;
6       for (v = 0;  v < n;  v++)
7           if ( flag[v] == 0)  bfs_visit (v) ;
}
```

BFS

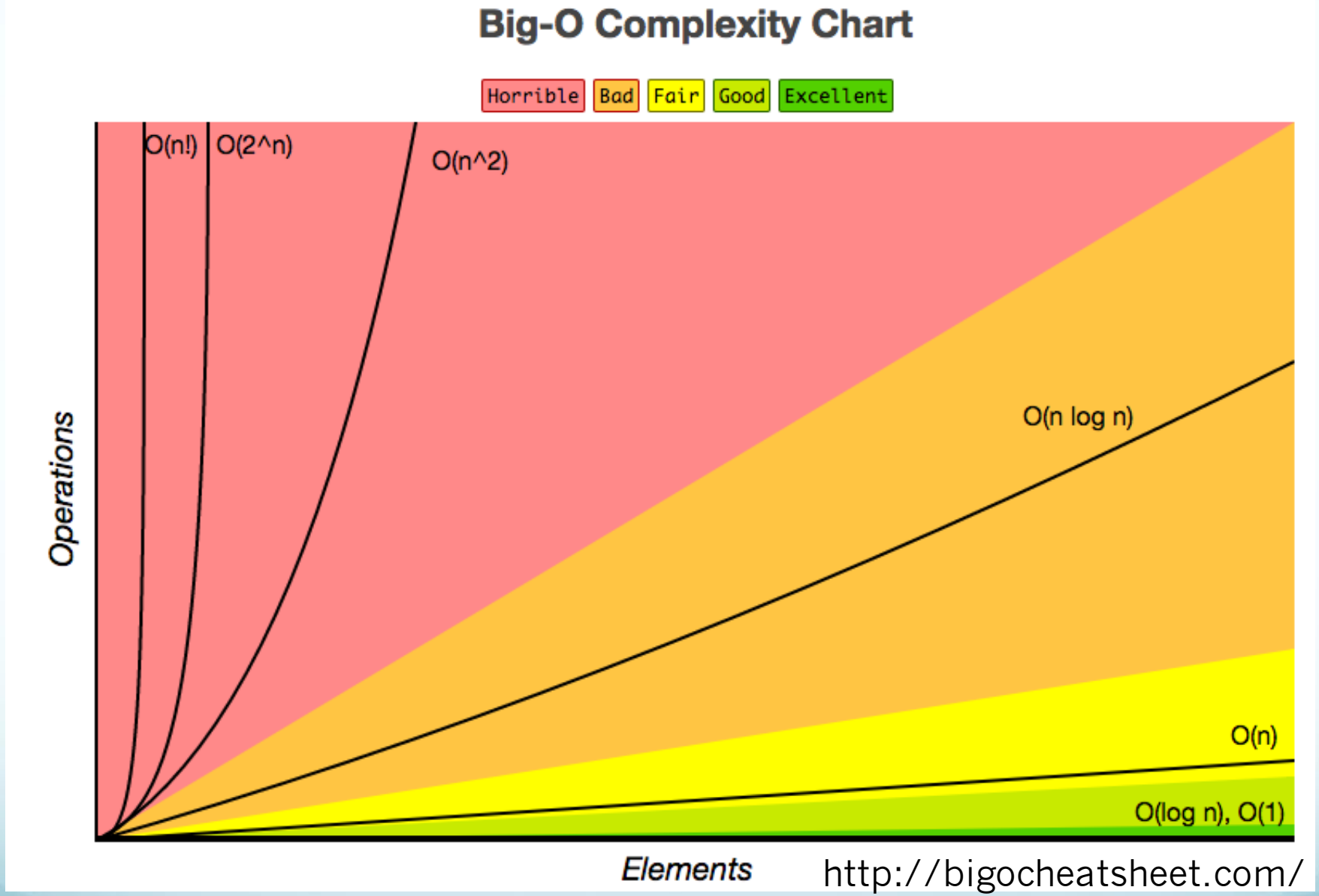
```
 bfs_visit (int v)
//입력 : 정점 번호 v
//출력 : v가 속한 연결 성분의 너비 우선 방문 순서를 flag 배열에.
   {
1       struct node  *t ;  int u ;
2       enqueue (v) ;  flag[v] = -1 ;
3       while (!queue_empty( ) )  {
4          u = delqueue ( );
5          mark ++ ;  flag[u] = mark ;
6          for (t = head[u];  t != NULL;  t -> next)
7             if (flag [ t->vertex] == 0)  {
8                enqueue ( t-> vertex) ;
9                flag[t->vertex] =- 1 ;
10            }
11      }
    }
```

시간 효율성 평가

빅오 표기법 (Big–O Notation)

- O(1) : 상수형, constant
- O(logn) : 로그형, logarithmic
- O(n) : 선형, linear
- O(nlogn) : 로그선형
- $O(n^2)$: 2차형, quadratic
- $O(n^3)$: 3차형, cubic
- $O(n^k)$: k차형, polynomial
- $O(2^n)$: 지수형, exponential
- O(n!) : 팩토리얼형

시간복잡도	*n*					
	1	2	4	8	16	32
1	1	1	1	1	1	1
Log n	0	1	2	3	4	5
n	1	2	4	8	16	32
n Log n	0	2	8	24	64	160
n^2	1	4	16	64	256	1024
n^3	1	8	64	512	4096	32768
2^n	2	4	16	256	65536	4294967296
n!	1	2	24	40326	20922789888000	26313×10^{33}



http://bigocheatsheet.com/

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < \cdots < O(2^n)$$

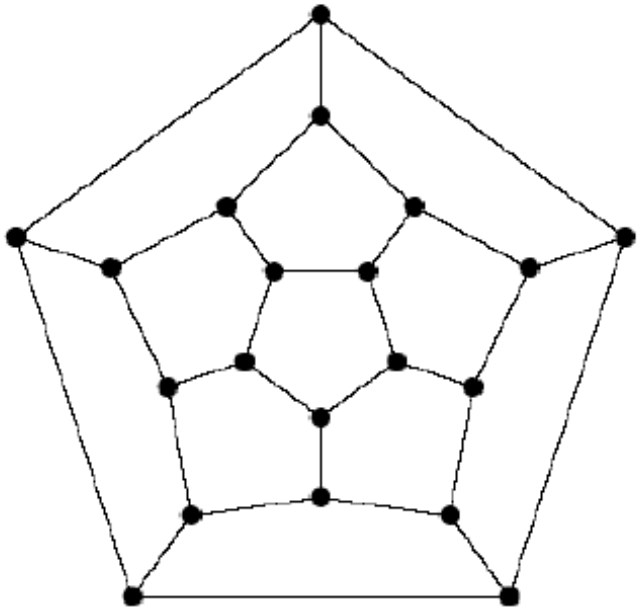
NP-완비성
NP-completeness

- 다항시간 알고리즘
  - 입력의 크기가 n 일 때 최악의 경우 어떤 상수 k에 대해 $O(n^k)$ 시간이 걸리는 알고리즘
  - 일반적으로 다항시간 알고리즘에 의해 풀 수 있는 문제를 "현실적인 시간에 풀 수 있는" 문제 또는 "쉬운" 문제로 생각하고 다항시간을 넘어서는 시간이 필요한 문제를 "현실적인 시간에 풀 수 없는" 또는 "어려운" 문제로 생각함.

- NP-complete problem
  - 상수 k에 대해 $O(n^k)$ 시간에 풀 수 없는 문제.
  - Somewhere between the polynomial problems, $O(n^k)$, and the exponential, $O(2^n)$, problems lines one particularly important category of problems called NP-complete problems.

Hamiltonian Cycle Problem

- Find a cycle that visits every vertex exactly once
- NP – complete



Game invented by Sir William Hamilton in 1857

Euler Cycle Problem

- Find a cycle that visits every edge exactly once
- Linear time

Find Eulerian Path

- Goal: Given an undirected or a directed graph, find a path or circuit that passes through each edge exactly once.
- Conditions

For an undirected graph

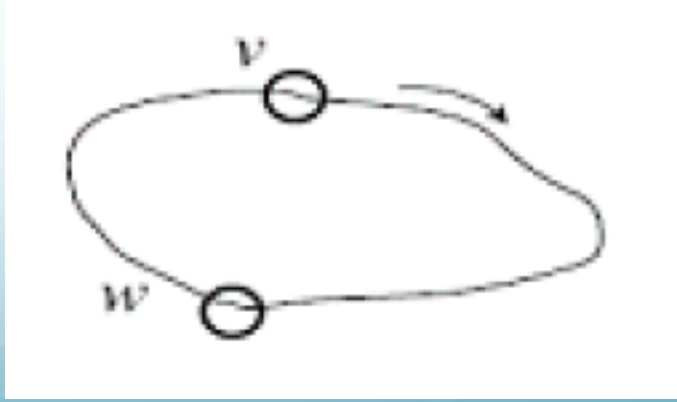
- An undirected graph has an eulerian circuit if and only if it is connected and each vertex has an even degree (degree is the number of edges that are adjacent to that vertex).
- An undirected graph has an eulerian path if and only if it is connected and all vertices except 2 have even degree. One of those 2 vertices that have an odd degree must be the start vertex, and the other one must be the end vertex.

For an directed graph

- A directed graph has an eulerian circuit if and only if it is connected and each vertex has the same in-degree as out-degree.
- A directed graph has an eulerian path if and only if it is connected and each vertex except 2 have the same in-degree as out-degree, and one of those 2 vertices has out-degree with one greater than in-degree (this is the start vertex), and the other vertex has in-degree with one greater than out-degree (this is the end vertex).

Eulerian Cycle 만드는 방법

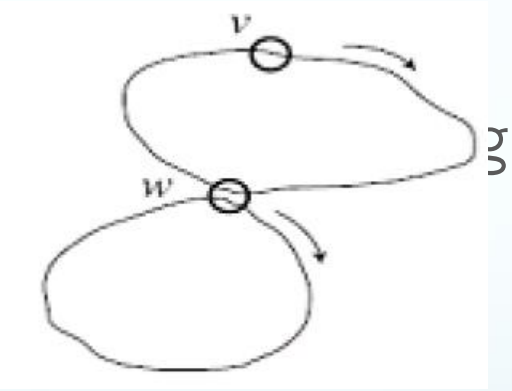
a. Start with an arbitrary vertex v and form an arbitrary cycle with unused edges until a dead end is reached. Since the graph is balanced, the only vertex where this can happen is the starting vertex v. This is because for any other vertex, the balanced condition ensures that for every incoming edge there is an outgoing edge that has not yet been used. Therefore, the resulting path will end at the same vertex where it started. i.e., vertex v.



Eulerian Cycle 만드는 방법

b. If the cycle from (a) is not an Eulerian cycle, it must contain a vertex w, which has untraversed edges. Perform previous step again, using vertex w as the starting point. Once again, we will end up in the starting vertex w.

c. Combine the cycles from (a) ar gle cycle and iterate step (b).



This algorithm can be implemented in linear time in the number of edges in the graph

Sequencing By Hybridization (SBH) Problem

- Reconstruct a string from its l-mer composition.
- Input : A set S, representing all l-mers from an unknown string s
- Output: String s such that Spectrum(s,l) = S

SBH Problem

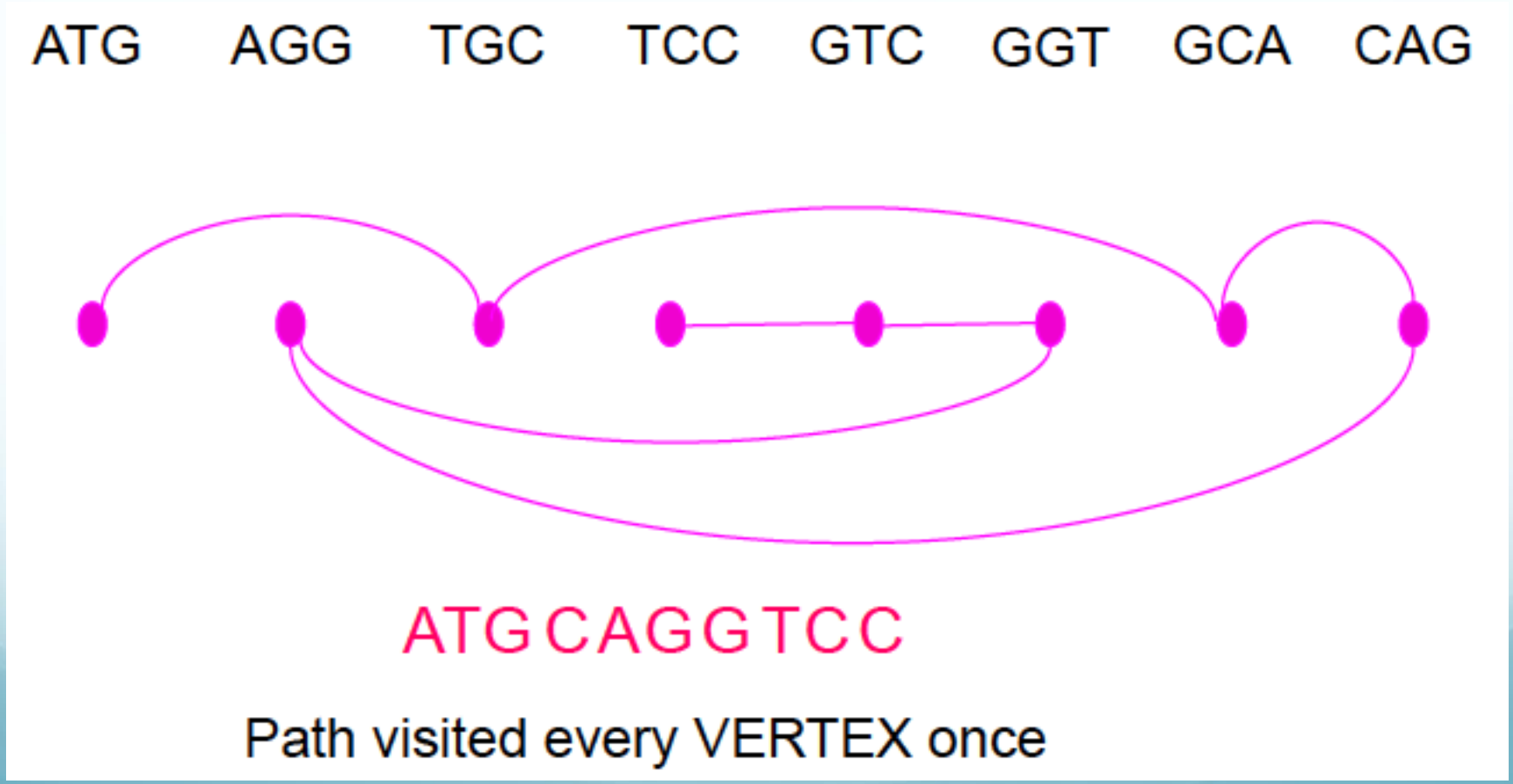
- Spectrum (s, l) - unordered multiset of all possible (n – l + 1) l-mers in a string s of length n
- The order of individual elements in Spectrum (s, l) does not matter
- For s = TATGGTGC all of the following are equivalent representations of Spectrum (s, 3):
  - {TAT, ATG, TGG, GGT, GTG, TGC}
  - {ATG, GGT, GTG, TAT, TGC, TGG}
  - {TGG, TGC, TAT, GTG, GGT, ATG}

Sequencing By Hybridization (SBH) Problem

- Reconstruct a string from its l-mer composition.
- Input : A set S, representing all l-mers from an unknown string s
- Output: String s such that Spectrum(s,l) = S

Hamiltonian Path Approach

- S = { ATG AGG TGC TCC GTC GGT GCA CAG }



Hamiltonian Path Approach

S = { ATG TGG TGC GTG GGC GCA GCG CGT }

Hamiltonian Path Approach

S = { ATG TGG TGC GTG GGC GCA GCG CGT }

Path 1:

?

Path 2:

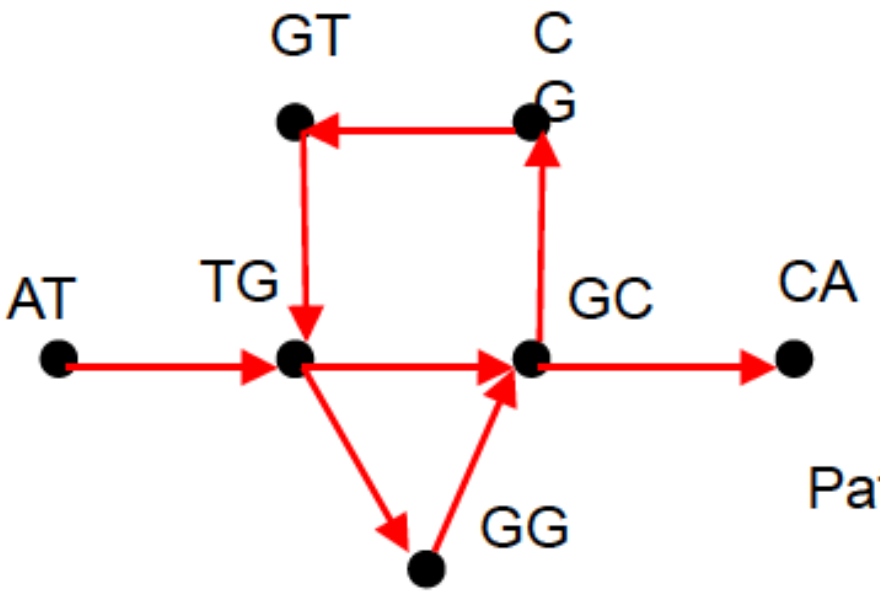
?

Euler Path

S = { ATG, TGC, GTG, GGC, GCA, GCG, CGT }

Vertices correspond to (l – 1) – mers : { AT, TG, GC, GG, GT, CA, CG }

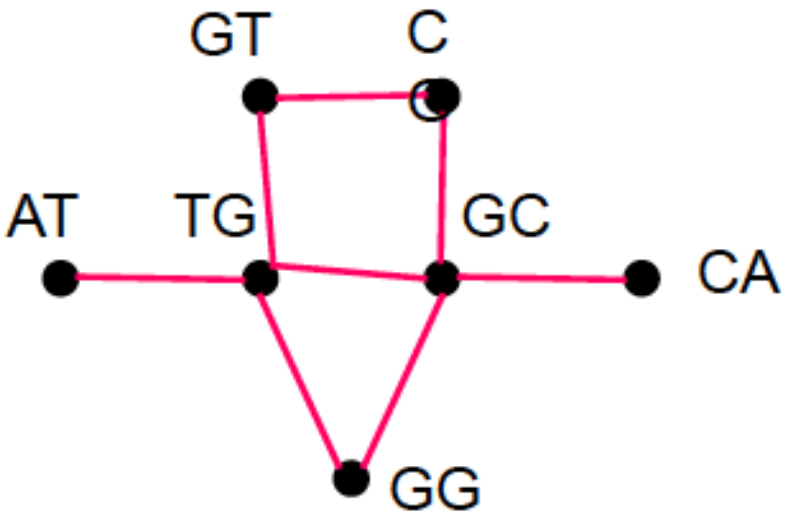
Edges correspond to l – mers from S



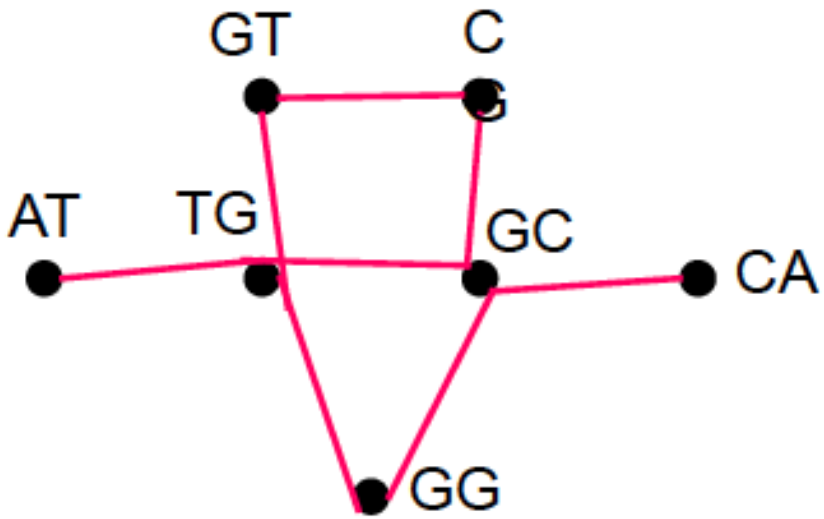
Path visited every EDGE once

Euler Path

S = { AT, TG, GC, GG, GT, CA, CG } corresponds to two different paths:



ATGGCGTGCA



ATGCGTGGCA

Hamiltonian path VS Eulerian path

- Notation
  - Input : A set S, representing all l-mers from an unknown string s
  - Output: String s such that Spectrum(s, l) = S
  - S = Spectrum(s, l)
  - $|S| = |s| - l + 1$
- Hamiltonian path
  - vertex: all l-mers
  - edge: direct edge(p,q) between p and q if last l-1 letters of p coincide with first l-2 letters of q
  - find a path that visits every "vertex" only once
  - time: construct graph + find path = $O(|S|^2) + O(2^{|s|}) = O(2^{|s|})$
  - NP-complete problem (between polynomial and exponential in worst case)
  - may not visit every edge

Hamiltonian path VS Eulerian path

- Notation
  - Input : A set S, representing all l-mers from an unknown string s
  - Output: String s such that Spectrum(s, l) = S
  - S = Spectrum(s, l)
  - |S| = |s|- l + 1
- Eulerian path
  - vertex: all (l-1)-mers
  - edge: direct edge(p,q) between p and q if spectrum(s,l) contains l-mers for which first l-1 letters coincide with p and last l-1 letters coincide with q
  - find a path that visits every “edge” only once
  - time: construct graph + find path = O(|S|) + O(|S|) = O(|S|), linear
  - may visit some vertices more than once

Summary of Eulerian path

- A connected graph is Eulerian if and only if each of its vertex is balanced (indegree=outdegree) except for start (1+indegree=outdegree) and end (indegree=a+outdegree) vertex.
- Find Eulerian cycle
  - Start from arbitrary vertex v, traverse univisited edges until meet a vertex which has no unvisited outgoing edges.
  - If the path is Eulerian, stop
  - Else it should contain a vertex w that has untraversed edges. Start from w, traverse unvisited edges until meet a vertex until meet a vertex which has no unvisited outgoing edge. compile two pathes (traverse the first path from v to w, traverse the second path from w to w itself, then traverse the remaining first path from w to v) : linear time algorithm.
  - Repeat the process above if there is any left edges.

references

- An Eulerian path approach to DNA fragment assembly, Pevzner et al, PNAS, 2011
- http://www.homolog.us/Tutorials/index.php?p=1.1&s=1
- http://www.cs.jhu.edu/~langmea/resources/lecture_notes/assembly_dbg.pdf