# Data structures for Graph

| | 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 |

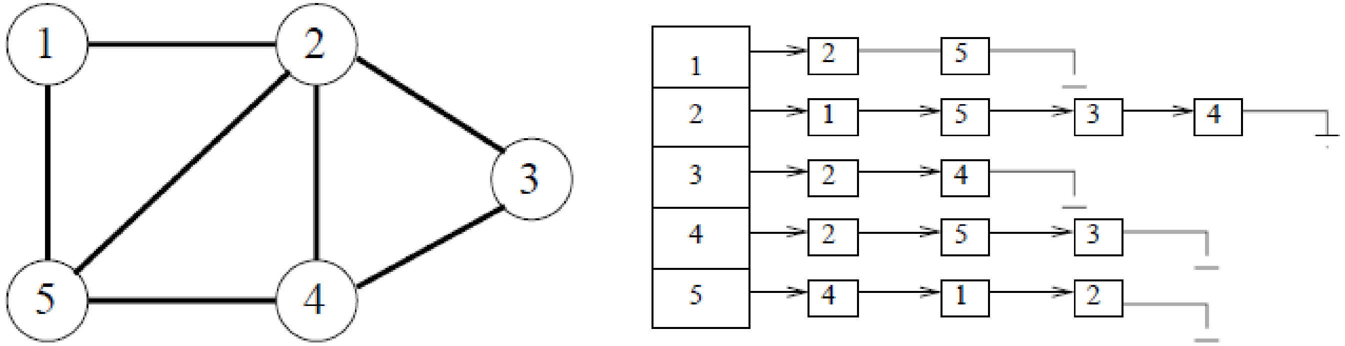



Figure 5.4: The adjacency matrix and adjacency list of a given graph

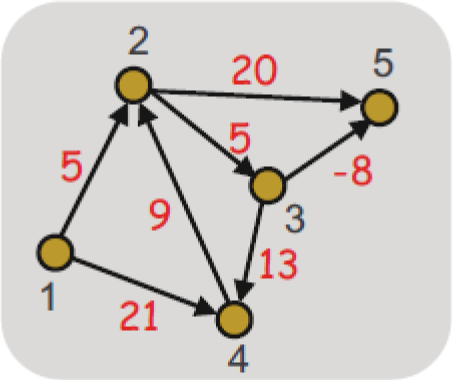
| Comparison | Winner |
|---|---|
| Faster to test if $(x, y)$ is in graph? | adjacency matrices |
| Faster to find the degree of a vertex? | adjacency lists |
| Less memory on small graphs? | adjacency lists $(m + n)$ vs. $(n^2)$ |
| Less memory on big graphs? | adjacency matrices (a small win) |
| Edge insertion or deletion? | adjacency matrices $O(1)$ vs. $O(d)$ |
| Faster to traverse the graph? | adjacency lists $\Theta(m + n)$ vs. $\Theta(n^2)$ |
| Better for most problems? | adjacency lists |

Figure 5.5: Relative advantages of adjacency lists and matrices.

# Data structures for Graph

❖ **adjacency matrix**

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 5 | 0 | 21 | 0 |
| 2 | 0 | 0 | 5 | 0 | 20 |
| 3 | 0 | 0 | 0 | 13 | -8 |
| 4 | 0 | 9 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 |




❖ **adjacency list**

1 → < (2,5),(4,21) >

2 → < (3,5),(5,20) >

3 → < (4,13),(5,-8) >

4 → < (2,9) >

5 → < >

# Data structures for Graph

```
#define MAXV            1000        /* maximum number of vertices */

typedef struct {
        int y;                      /* adjacency info */
        int weight;                 /* edge weight, if any */
        struct edgenode *next;      /* next edge in list */
} edgenode;


typedef struct {
        edgenode *edges[MAXV+1];    /* adjacency info */
        int degree[MAXV+1];         /* outdegree of each vertex */
        int nvertices;              /* number of vertices in graph */
        int nedges;                 /* number of edges in graph */
        bool directed;              /* is the graph directed? */
} graph;
```

# Data structures for Graph

```
initialize_graph(graph *g, bool directed)
{
        int i;                          /* counter */

        g -> nvertices = 0;
        g -> nedges = 0;
        g -> directed = directed;

        for (i=1; i<=MAXV; i++) g->degree[i] = 0;
        for (i=1; i<=MAXV; i++) g->edges[i] = NULL;
}
```

# Data structures for Graph

```
read_graph(graph *g, bool directed)
{
        int i;                              /* counter */
        int m;                              /* number of edges */
        int x, y;                           /* vertices in edge (x,y) */

        initialize_graph(g, directed);

        scanf("%d %d",&(g->nvertices),&m);

        for (i=1; i<=m; i++) {
                scanf("%d %d",&x,&y);
                insert_edge(g,x,y,directed);
        }
}
```

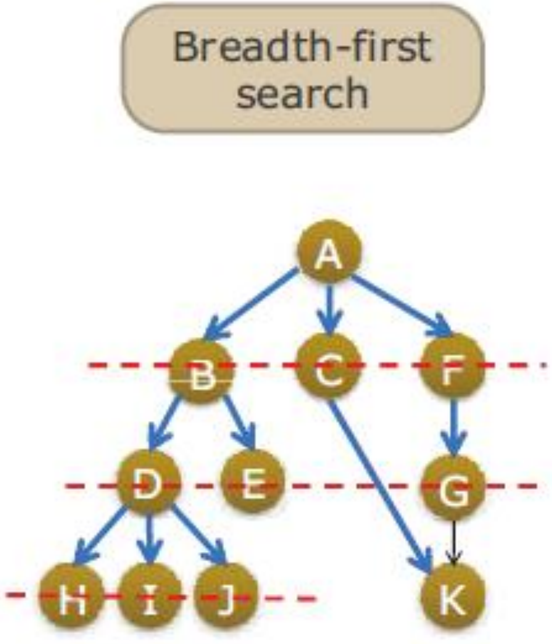
# Data structures for Graph

```
insert_edge(graph *g, int x, int y, bool directed)
{
      edgenode *p;                    /* temporary pointer */

      p = malloc(sizeof(edgenode)); /* allocate edgenode storage */

      p->weight = NULL;
      p->y = y;
      p->next = g->edges[x];

      g->edges[x] = p;               /* insert at head of list */

      g->degree[x] ++;

      if (directed == FALSE)
              insert_edge(g,y,x,TRUE);
      else
              g->nedges ++;
}
```

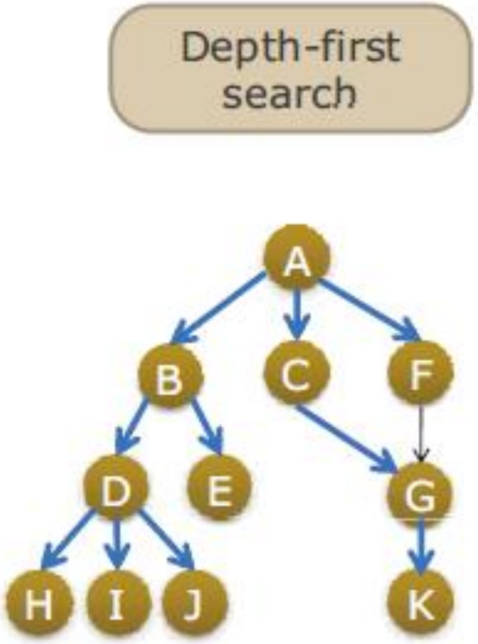
# Data structures for Graph

```
print_graph(graph *g)
{
        int i;                          /* counter */
        edgenode *p;                    /* temporary pointer */

        for (i=1; i<=g->nvertices; i++) {
                printf("%d: ",i);
                p = g->edges[i];
                while (p != NULL) {
                        printf(" %d",p->y);
                        p = p->next;
                }
                printf("\n");
        }
}
```

# Graph Traversal



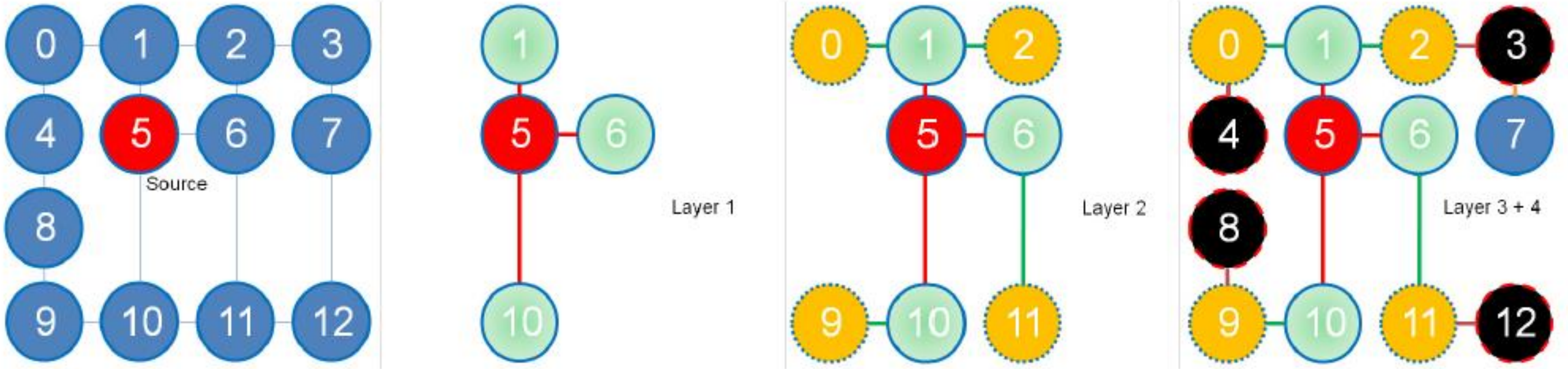

ใช้ queue จำปม
ระหว่างการค้น




ใช้ stack จำปม
ระหว่างการค้น

# Graph Traversal: Breath First Search

- BFS will visit vertices that are direct neighbors of the source vertex (first layer), neighbors of direct neighbors (second layer), and so on, layer by layer.
- BFS algorithm also runs in $O(V+E)$ and $O(V^2)$

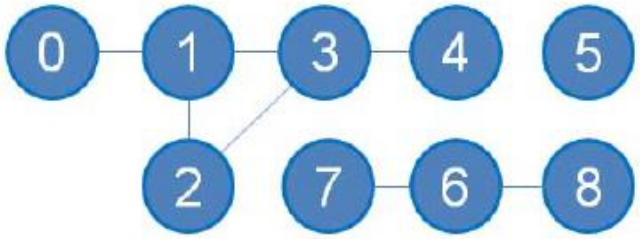


# Graph Traversal: Breath First Search

```
// inside int main()---no recursion
  vi d(V, INF); d[s] = 0;                      // distance from source s to s is 0
  queue<int> q; q.push(s);                                    // start from source

  while (!q.empty()) {
    int u = q.front(); q.pop();                              // queue: layer by layer!
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];                                 // for each neighbor of u
      if (d[v.first] == INF) {               // if v.first is unvisited + reachable
        d[v.first] = d[u] + 1;                 // make d[v.first] != INF to flag it
        q.push(v.first);                   // enqueue v.first for the next iteration
  } } }
```

# Graph Traversal: Depth First Search

- In a graph with *V* vertices and *E* edges, DFS runs in *O*(*V* + *E*) and *O*(*V* 2) if the graph is stored as Adjacency List and Adjacency Matrix, respectively




- dfs(0)—calling DFS from a starting vertex *u* = 0—will trigger this sequence of visitation: 0 → 1 → 2 → 3 → 4. the sequence 0 → 1 → 3 → 2 (backtrack to 3) → 4 is also a possible visitation sequence.
- Also notice that one call of dfs(u) will only visit all vertices that are *connected* to vertex *u*. That is why vertices 5, 6, 7, and 8 remain unvisited after calling dfs(0).

# Graph Traversal: Depth First Search

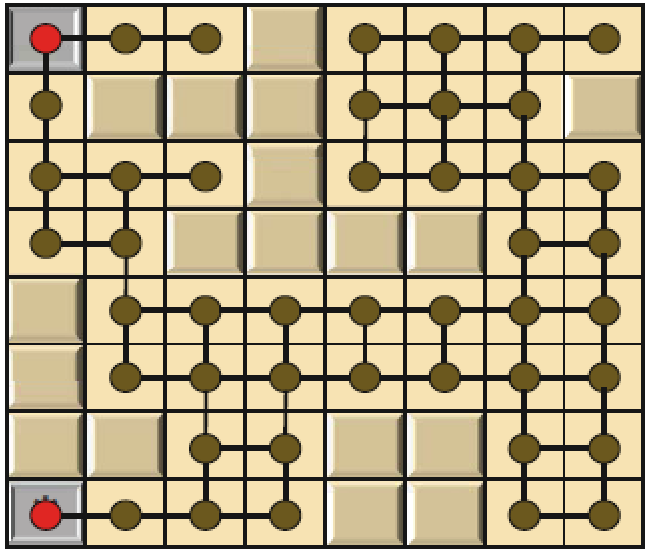
```
typedef pair<int, int> ii; // In this chapter, we will frequently use these
typedef vector<ii> vii; // three data type shortcuts. They may look cryptic
typedef vector<int> vi;   // but they are useful in competitive programming

vi dfs_num;   // global variable, initially all values are set to UNVISITED

void dfs(int u) {     // DFS for normal usage: as graph traversal algorithm
  dfs_num[u] = VISITED;         // important: we mark this vertex as visited
  for (int j = 0; j < (int)AdjList[u].size(); j++) { // default DS: AdjList
    ii v = AdjList[u][j];                       // v is a (neighbor, weight) pair
    if (dfs_num[v.first] == UNVISITED)    // important check to avoid cycle
      dfs(v.first);    // recursively visits unvisited neighbors of vertex u
} }  // for simple graph traversal, we ignore the weight stored at v.second
```

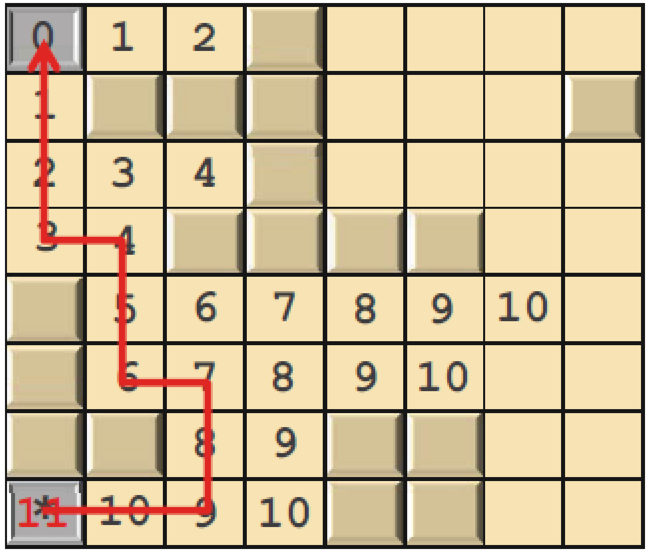
# Graph Traversal: BFS or DFS

- Shortest path
- Finding cycle in graph
- Connected components
- Two-coloring graphs, Bipartite graphs
- UVa 00429 - Word Transformation
- UVa 11902 – Dominator
- 15-puzzle

# Graph Traversal

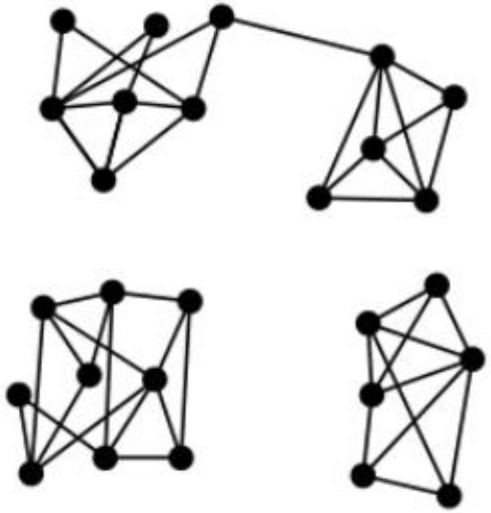
- Shortest path

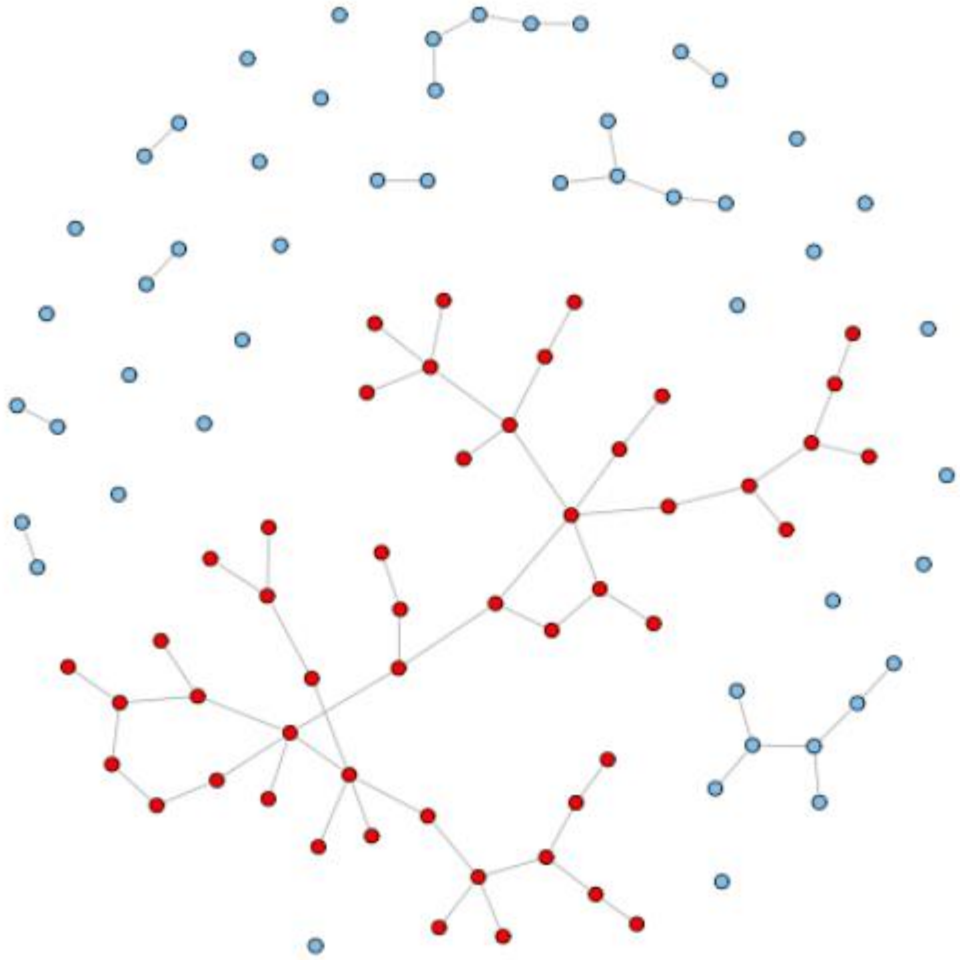




# Graph Traversal

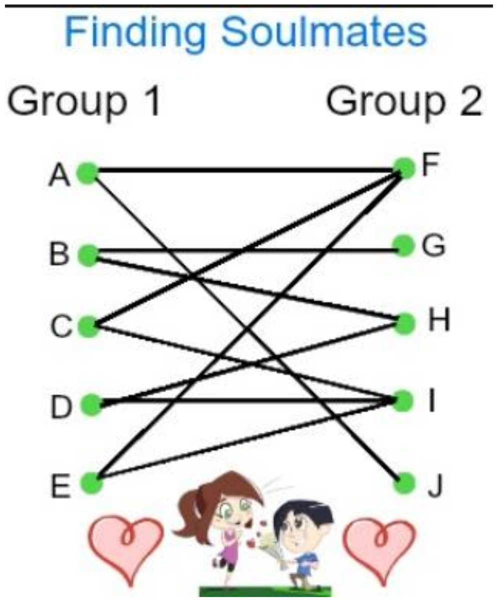
- Connected component

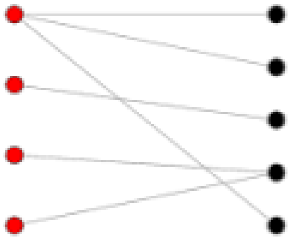


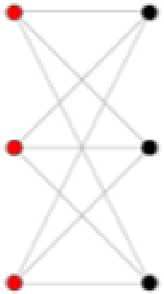


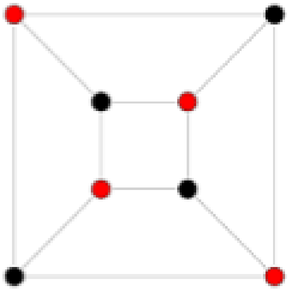
# Graph Traversal

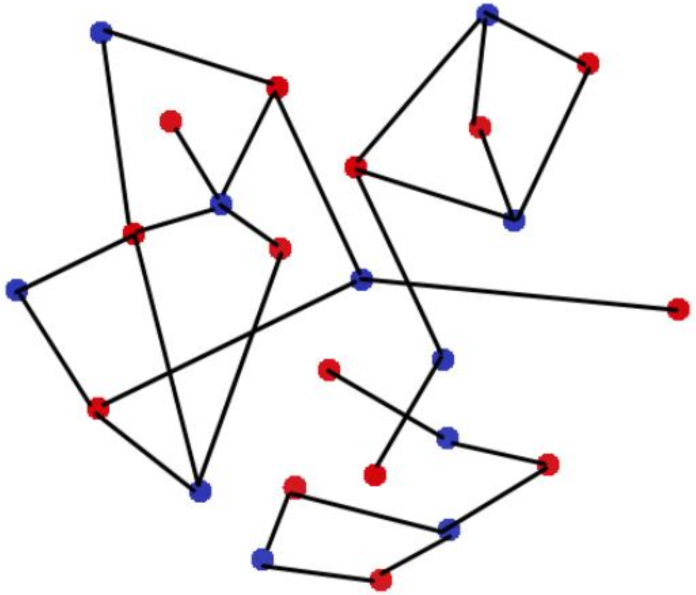
- Bipartite graphs



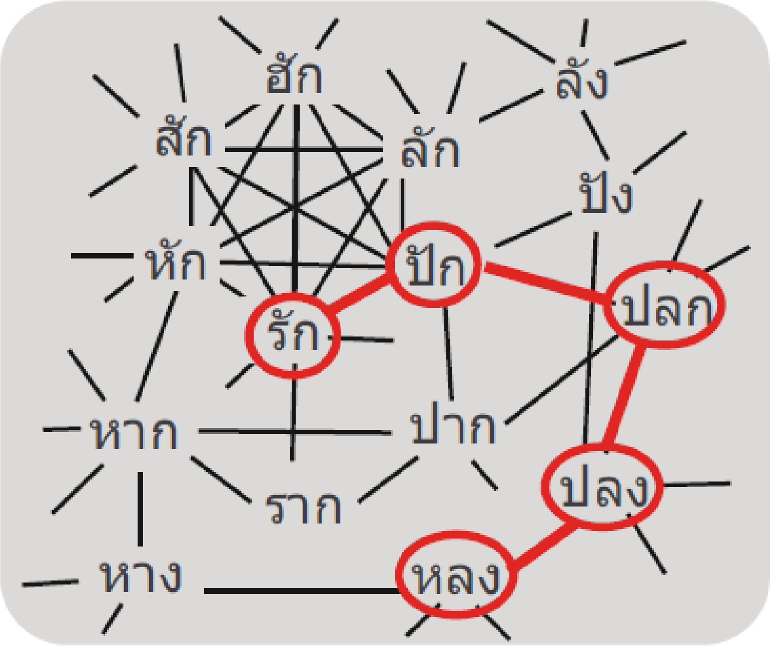






# Graph Traversal

- UVa 00429 - Word Transformation
  - each word is a vertex, connect 2 words with an edge if differ by 1 letter

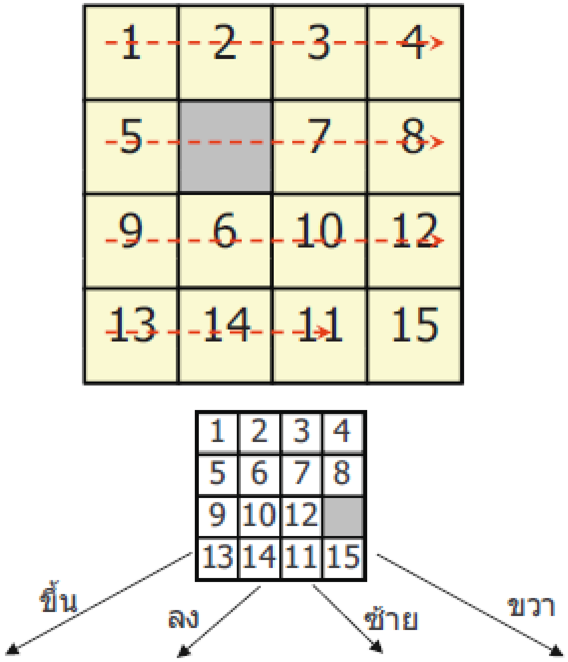


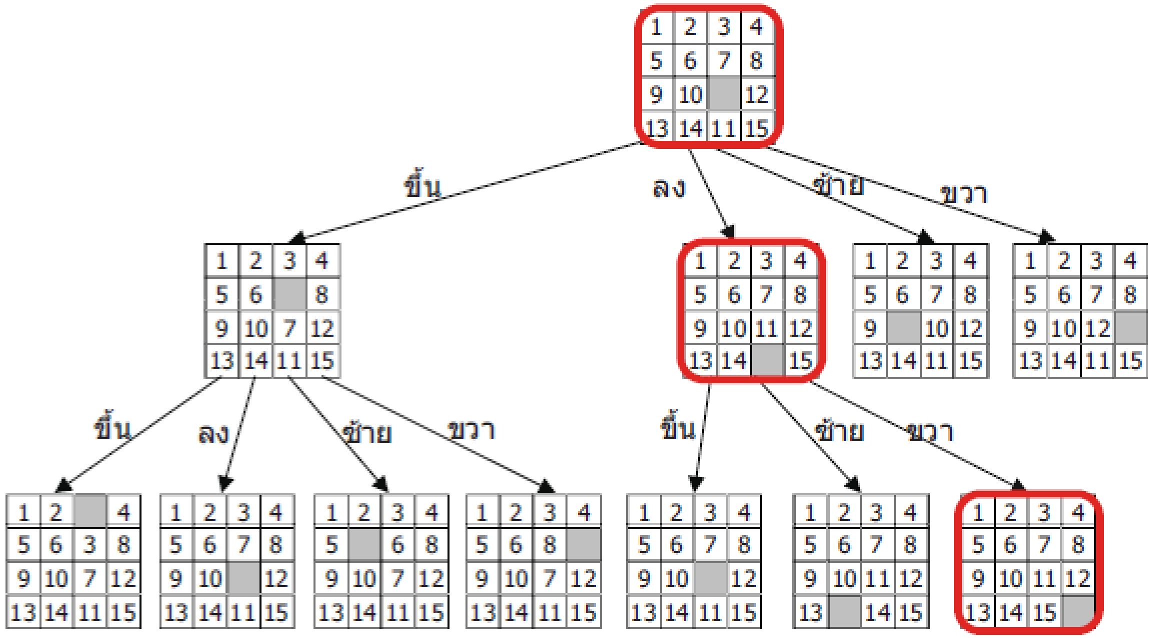
# Graph Traversal

- UVa 11902 – Dominator
  - Reachability test
  - $O(V \times V^2 = V^3)$ algorithm.
  - Run dfs(0) on the input graph to record vertices that are reachable from vertex 0.
  - Then to check which vertices are dominated by vertex $X$, we (temporarily) turn off all the outgoing edges of vertex $X$ and rerun dfs(0).
  - Now, a vertex $Y$ is not dominated by vertex $X$ if dfs(0) initially cannot reach vertex $Y$ or dfs(0) can reach vertex $Y$ even after all outgoing edges of vertex $X$ are (temporarily) turned off. Vertex $Y$ is dominated by vertex $X$ otherwise.
  - We repeat this process $\forall X \in [0 \ldots V - 1]$.
  - Tips: We do not have to physically delete vertex $X$ from the input graph. We can simply add a statement inside our DFS routine to stop the traversal if it hits vertex $X$.

# Graph Traversal

- 15-puzzle



# Priority Queue in C++

```cpp
#include <functional>
#include <queue>
#include <vector>
#include <iostream>

template<typename T> void print_queue(T& q) {
    while(!q.empty()) {
        std::cout << q.top() << " ";
        q.pop();
    }
    std::cout << '\n';
}

int main() {
    std::priority_queue<int> q;

    for(int n : {1,8,5,6,3,4,0,9,7,2})
        q.push(n);

    print_queue(q);

    std::priority_queue<int, std::vector<int>, std::greater<int> > q2;

    for(int n : {1,8,5,6,3,4,0,9,7,2})
        q2.push(n);

    print_queue(q2);

    // Using lambda to compare elements.
    auto cmp = [](int left, int right) { return (left ^ 1) < (right ^ 1);};
    std::priority_queue<int, std::vector<int>, decltype(cmp)> q3(cmp);

    for(int n : {1,8,5,6,3,4,0,9,7,2})
        q3.push(n);

    print_queue(q3);

}
```

Output:

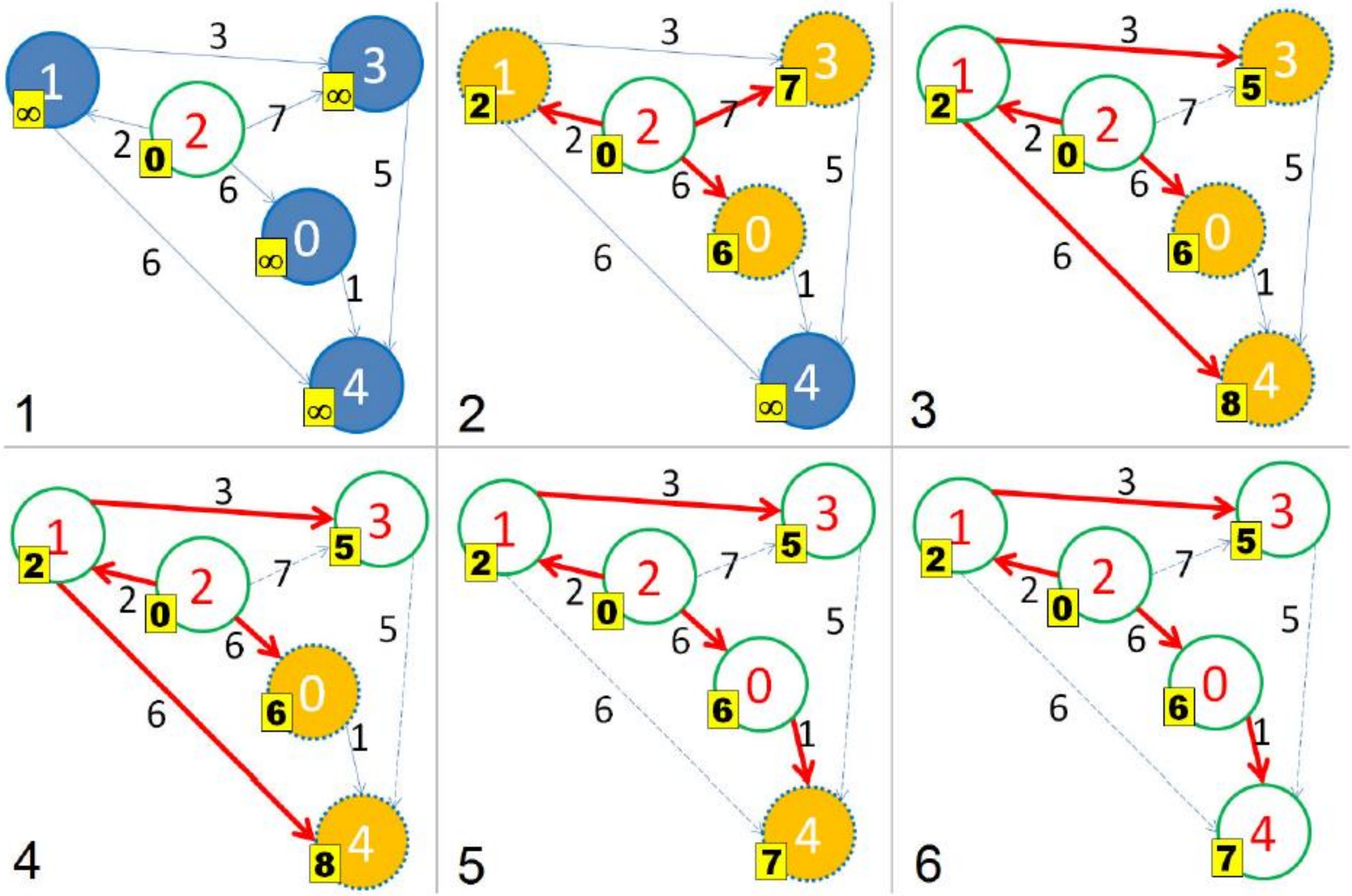
```
9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8 9
8 9 6 7 4 5 2 3 0 1
```

# Single source shortest path
# Dijkstra’s algorithm

$O((V+E)\log V)$ Max Size: V, E < 300,000

```
vi dist(V, INF); dist[s] = 0;                    // INF = 1B to avoid overflow
priority_queue< ii, vector<ii>, greater<ii> > pq; pq.push(ii(0, s));
while (!pq.empty()) {                                            // main loop
  ii front = pq.top(); pq.pop(); // greedy: get shortest unvisited vertex
  int d = front.first, u = front.second;
  if (d > dist[u]) continue;                  // this is a very important check
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];                         // all outgoing edges from u
    if (dist[u] + v.second < dist[v.first]) {
      dist[v.first] = dist[u] + v.second;                 // relax operation
      pq.push(ii(dist[v.first], v.first));
} } }     // this variant can cause duplicate items in the priority queue
```

# Single source shortest path
# Dijkstra’s algorithm




$\{(0,2)\}$

$\{(2,1),\ (6,0),\ (7,3)\}.$

$\{\underline{(5,3)},\ (6,0),\ \underline{(7,3)},\ (8,4)\}$

$\{(6,0),\ (7,3),\ (8,4)\}.$

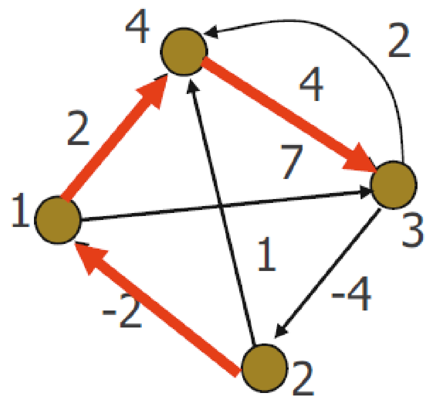
$\{(7,3),\ \underline{(7,4)},\ \underline{(8,4)}\}$

# All pair shortest path: Floyd Warshall

**กราฟ G (V, E) เส้นเชื่อมมีน้ำหนัก มีทิศทาง**

เส้นเชื่อมยาวเป็นลบได้

แต่ต้องไม่มีวงที่ความยาวรวมเป็นลบ



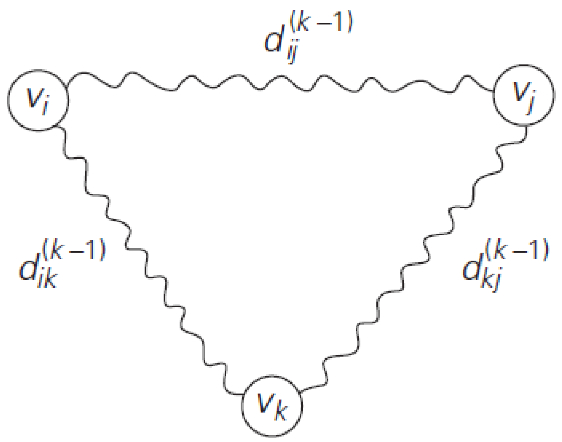

shortest path tree ของวิถีสั้นสุดจากปม 2 ถึงปมอื่น ๆ

| $W$ | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | – | 7 | 2 |
| 2 | -2 | 0 | – | 1 |
| 3 | – | -4 | 0 | 2 |
| 4 | – | – | 4 | 0 |

| $D$ | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 2 | 6 | 2 |
| 2 | -2 | 0 | 4 | 0 |
| 3 | -6 | -4 | 0 | -4 |
| 4 | -2 | 0 | 4 | 0 |

อ. สมชาย ประสิทธิ์จูตระกูล

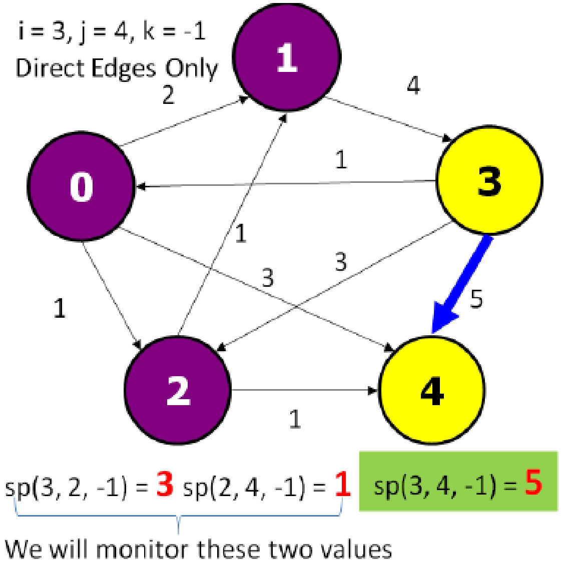
# All pair shortest path
# Floyd Warshall




$O(V^3)$ Max Size: V < 400

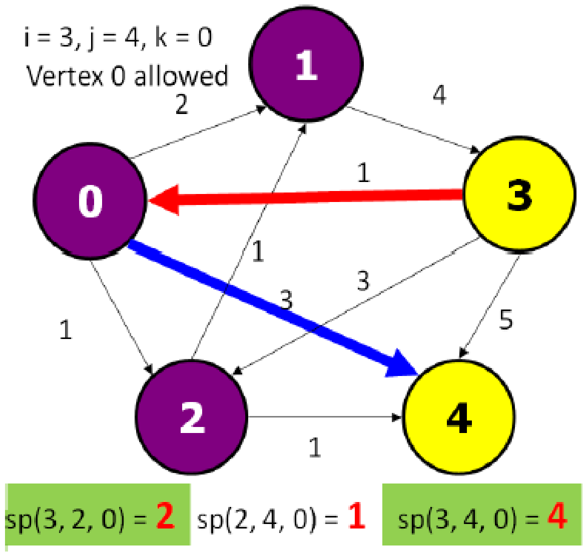
```
// inside int main()
  // precondition: AdjMat[i][j] contains the weight of edge (i, j)
  // or INF (1B) if there is no such edge
  // AdjMat is a 32-bit signed integer array
  for (int k = 0; k < V; k++)          // remember that loop order is k->i->j
    for (int i = 0; i < V; i++)
      for (int j = 0; j < V; j++)
        AdjMat[i][j] = min(AdjMat[i][j], AdjMat[i][k] + AdjMat[k][j]);
```

# All pair shortest path: Floyd Warshall




The current content of Adjacency Matrix D
at **k = -1**

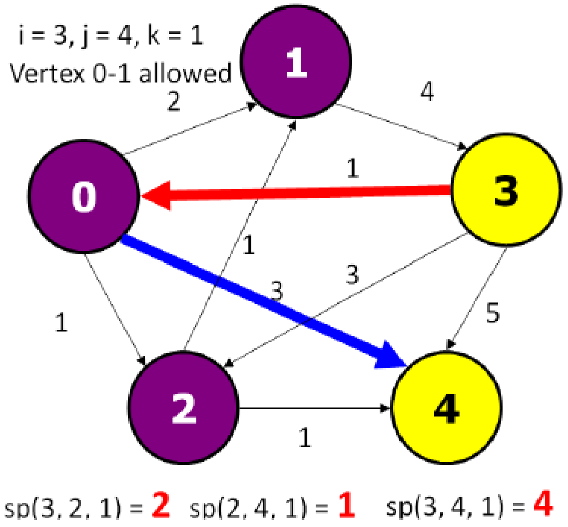
| k = -1 | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 2 | 1 | ∞ | 3 |
| 1 | ∞ | 0 | ∞ | 4 | ∞ |
| 2 | ∞ | 1 | 0 | ∞ | 1 |
| 3 | 1 | ∞ | 3 | 0 | **5** |
| 4 | ∞ | ∞ | ∞ | ∞ | 0 |




The current content of Adjacency Matrix D
at **k = 0**

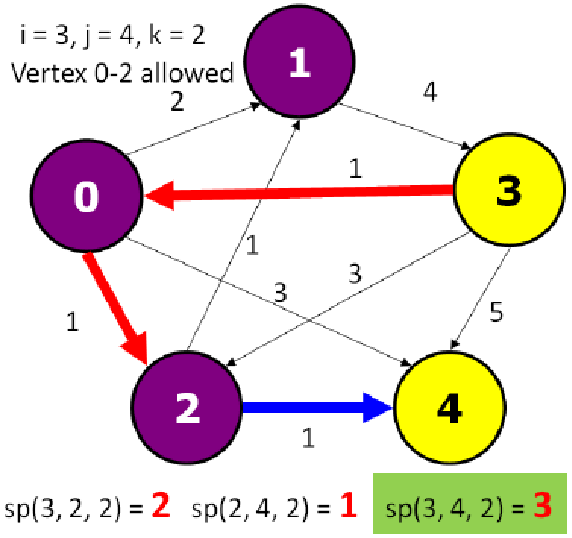
| k = 0 | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 2 | 1 | ∞ | 3 |
| 1 | ∞ | 0 | ∞ | 4 | ∞ |
| 2 | ∞ | 1 | 0 | ∞ | 1 |
| 3 | 1 | 3 | 2 | 0 | 4 |
| 4 | ∞ | ∞ | ∞ | ∞ | 0 |

# All pair shortest path: Floyd Warshall




The current content of Adjacency Matrix D at **k = 1**

| k = 1 | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 2 | 1 | 6 | 3 |
| 1 | ∞ | 0 | ∞ | 4 | ∞ |
| 2 | ∞ | 1 | 0 | 5 | 1 |
| 3 | 1 | 3 | 2 | 0 | 4 |
| 4 | ∞ | ∞ | ∞ | ∞ | 0 |




The current content of Adjacency Matrix D at **k = 2**

| k = 2 | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 2 | 1 | 6 | 2 |
| 1 | ∞ | 0 | ∞ | 4 | ∞ |
| 2 | ∞ | 1 | 0 | 5 | 1 |
| 3 | 1 | 3 | 2 | 0 | 3 |
| 4 | ∞ | ∞ | ∞ | ∞ | 0 |

# Problems

- UVa 01112 - Mice and Maze
- UVa 11463 – Commandos
- UVa 00104 – Arbitrage
- UVa 11367 - Full Tank

- บางทีโจทย์จะไม่ได้ให้เรา run Algo… บนกราฟธรรมดาๆที่ให้มาตรงๆ
  1. นำปัญหาที่ตอนแรกอาจดูไม่เหมือนเกี่ยวกับกราฟ มาแปลงเป็นรูปแบบกราฟ, อะไรคือ vertex? อะไรคือ edge?
  2. โจทย์ให้กราฟมา แต่ที่จริงยังทำ Algo… เลยไม่ได้ ต้องแปลงกราฟที่ให้มาเป็นกราฟอีกรูปแบบนึงก่อนแล้วค่อยทำ Algo… บนกราฟนั้น

- UVa 00104 – Arbitrage
  - Detecting negative cycle in Floyd Warshall
    - Negative Cycle can be identified by looking at the diagonals of the dist[][] matrix generated by Floyd-Warshall algorithm.
    - diagonal dist[2][2] value is smaller than 0 means, a path starting from 2 and ending at 2 results in a negative cycle
    - you are looking for a cycle whose product of edge weights is >1, i.e. w1 * w2 * w3 * ... > 1.
    - $\log (w_1 * w_2 * w_3 \ldots) > \log(1)$
    - $\log(w_1) + \log(w_2) + \log(w_3) \ldots > 0$
    - $-\log(w_1) - \log(w_2) - \log(w_3) \ldots < 0$
  - Path reconstruction
    - Create another table/matrix p
    - p[i][j] store the predecessor node k that was chosen in the inner loop of Floyd algorithm

- UVa 11367 - Full Tank
  - Dijkstra on modify graph,
  - State-Space graph
    - (location, fuel), new#V = #V x fuel capacity
    - 2 type of edge
      - (x,fuel) – cost -> (x, fuel + 1)
      - (x,fuel) – 0 -> (y, fuel – length(x,y))
    - Start state (s,0)
    - End stage (e, any)