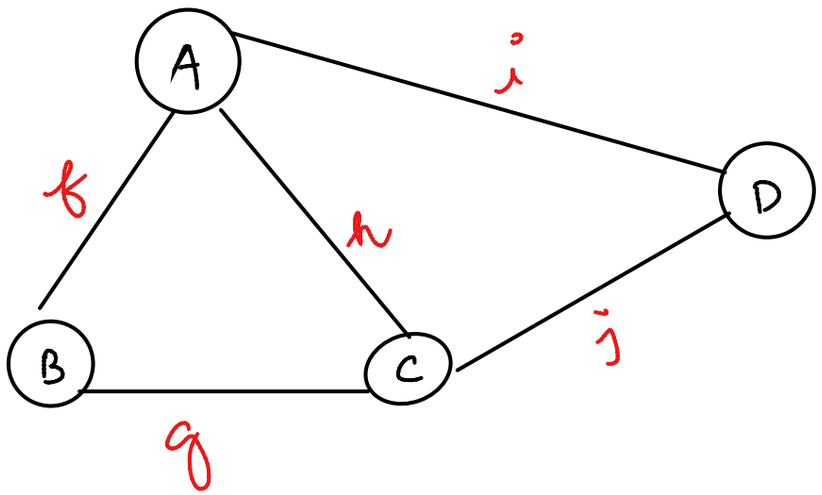
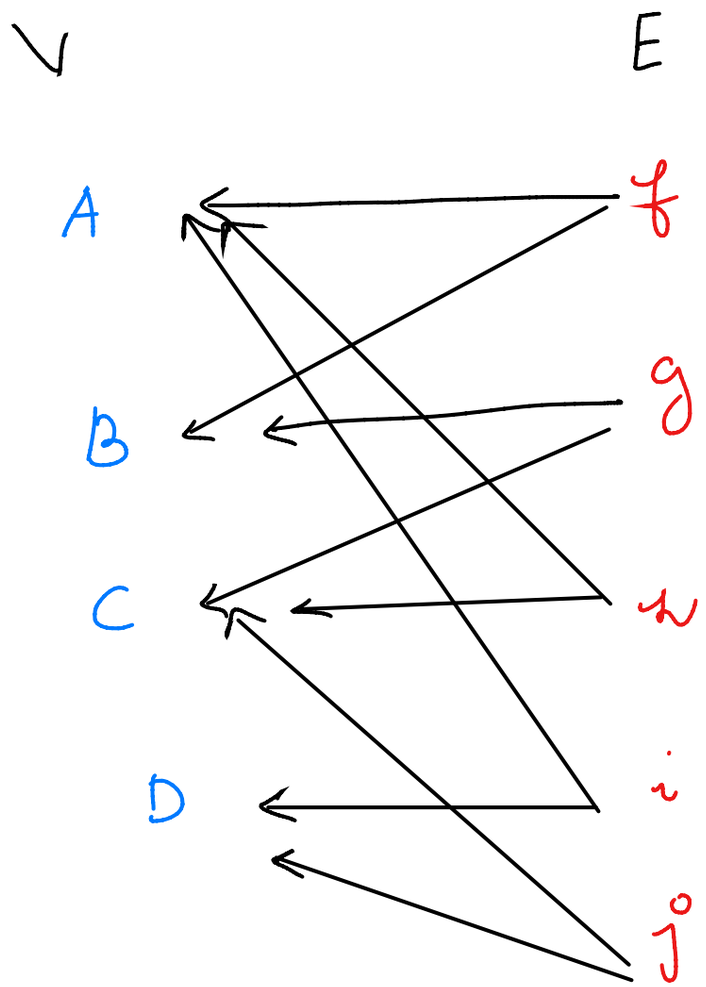
Edge List





vertex_count(), edge_count()

vertices()

edges()

get_edge(A, B), remove_edge()

add_vertex(V), add_edge(e, V, v)

remove_vertex()

degree_vertex(), incident_edges

O(1)

O(n) → # vertices

O(m) → # edges

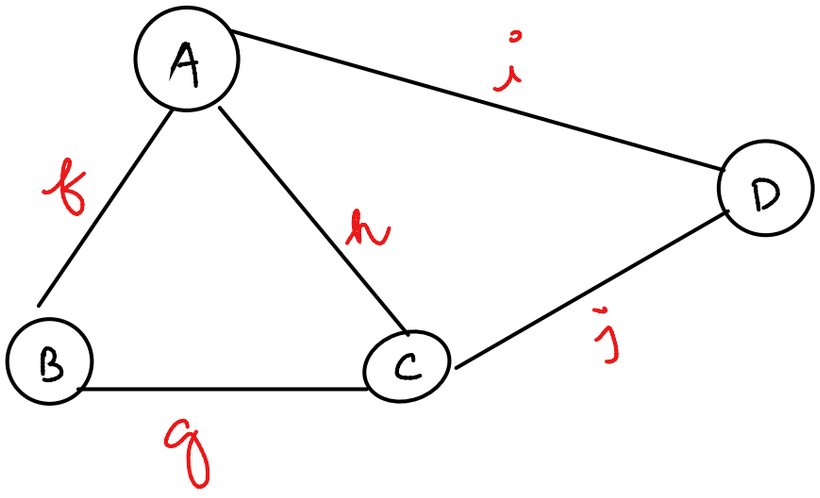
O(m)

O(1)

O(n)

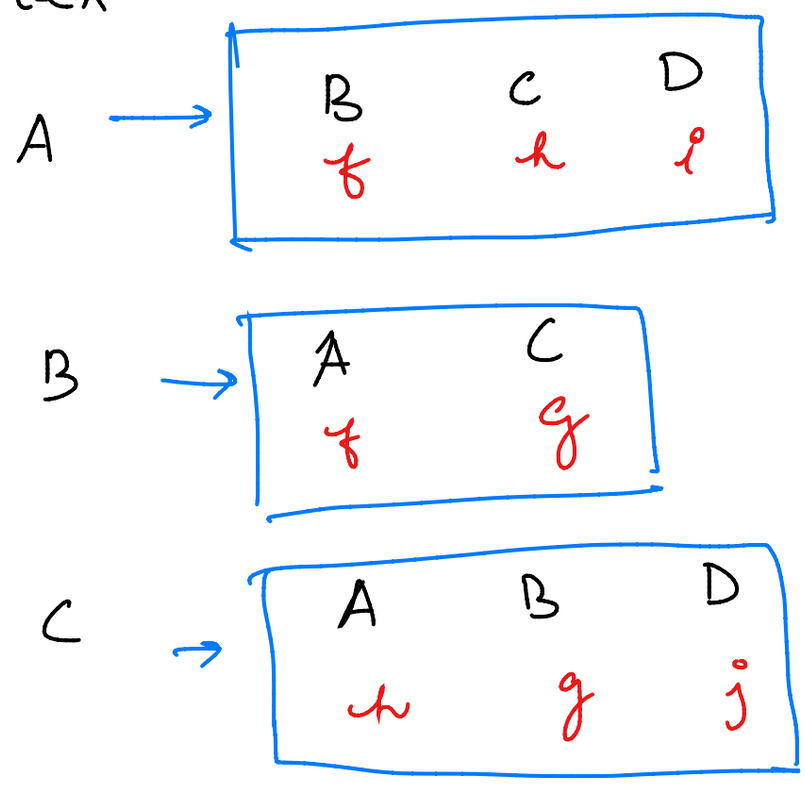
O(m)

Adjacency List



Adjacency list

Vertex

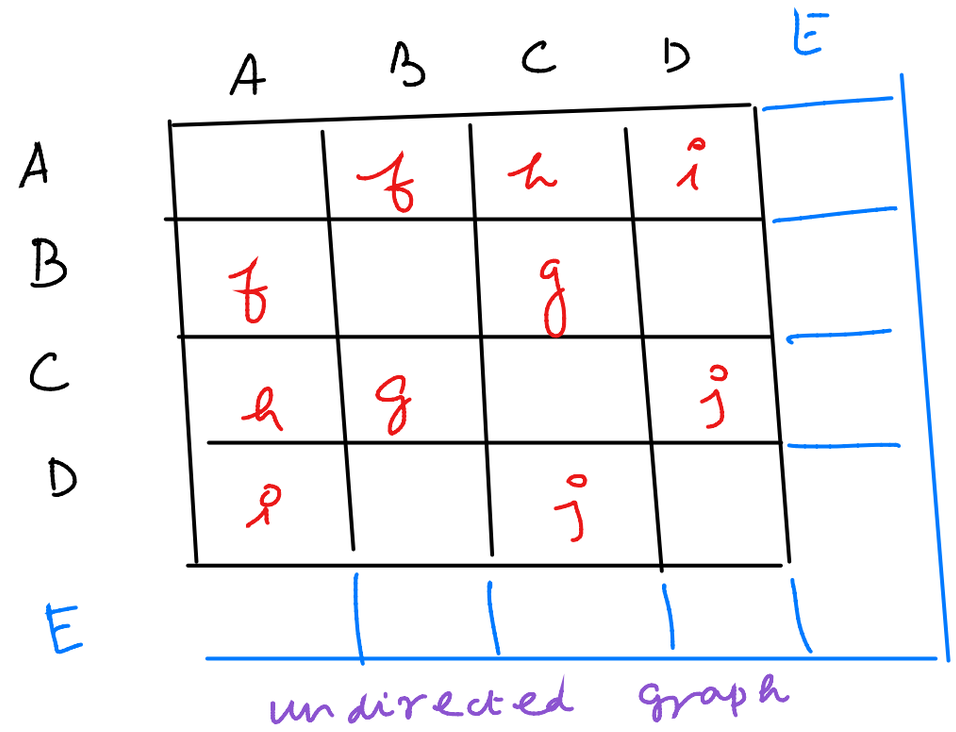


Get_edge (A, B) = $O(\min(d_A, d_B))$

Adjacency map instead of list

Get_edge (A, B) = $O(1)$

Adjacenly Matrix $n \times n$ matrix



get-edge (A, B) = O(1)

If we want to add extra vertices then we have to

spend n^2 new operations to copy the matrix.

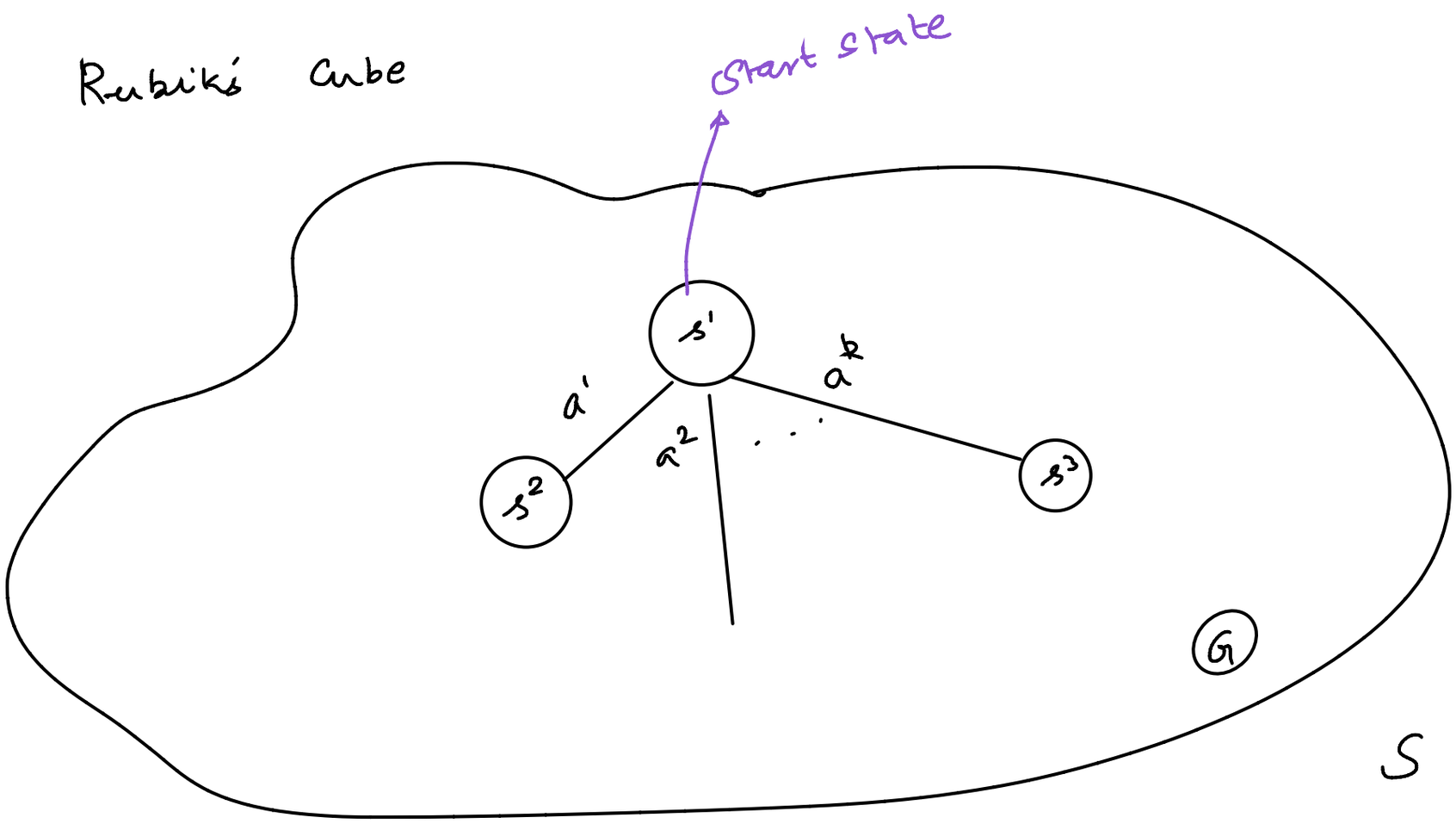
Graph Search

Breadth First , Depth First , Least Cost First , Heuristic

Applies to

- Graph Search
- Deterministic State Space search

Rubik's Cube



Goal

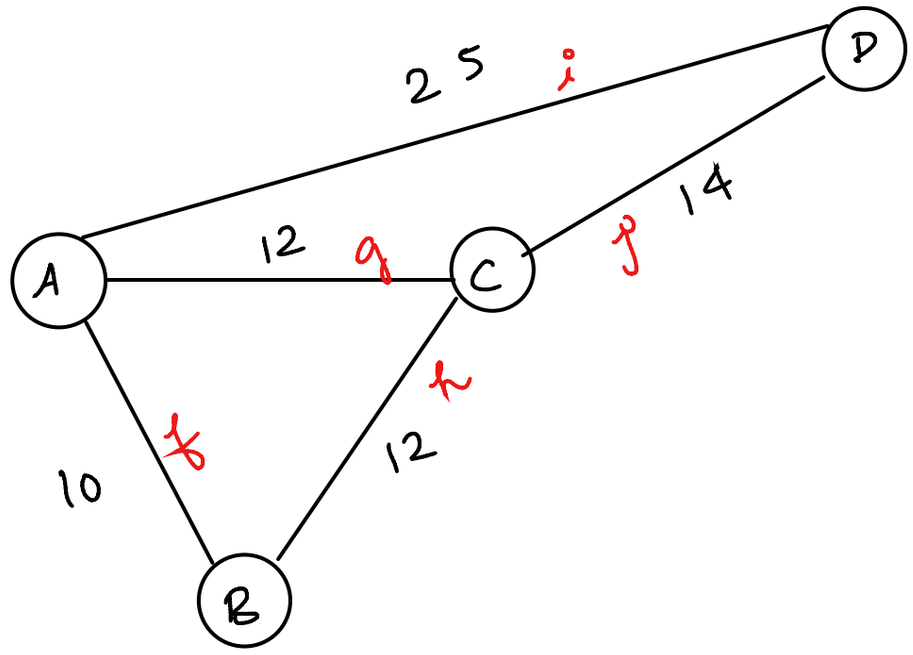
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	////

empty square

Start State

1	3	6	4
5	9	2	8
9	4	7	12
13	14	15	////

Road Network:



Costs are not same across edges.

First consider cost of edge to be 1.

Problem: Starting from start state (or start vertex) S we want to reach the Goal State G (or goal vertex)

Assume: unit step cost or edge cost = 1.

Data Structure :

- Search Tree
- Frontier (Implemented as a Priority Queue)
  Queue / Stack
- Explored List (optional)

Node in Search Tree

* vertex / State : n. STATE / n. vertex
* parent : n. Parent
* action / edge : n. Action / n. Edge
* Path cost : n. COST

Tree Search : No explored list

Graph Search : use explored list

Tree_Search

* Frontier = { start state / vertex }

* do

  * If Frontier = Empty ⇒ Search failed

  * Pick a node ∈ Frontier , Frontier = Frontier - Node

  * If node = Goal the stop (trace route back to start)

  * Frontier = Frontier + Children(node)