Graph

Graph Searches

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Search Algorithms

- Visit of a graph G=(V, E)
  - Starts from a given node
  - Follows the edges according to a known strategy
  - Lists the nodes found, possibly adding additional information
- Algorithms
  - Depth-First Search (DFS)
  - Breadth-First Search (BFS)

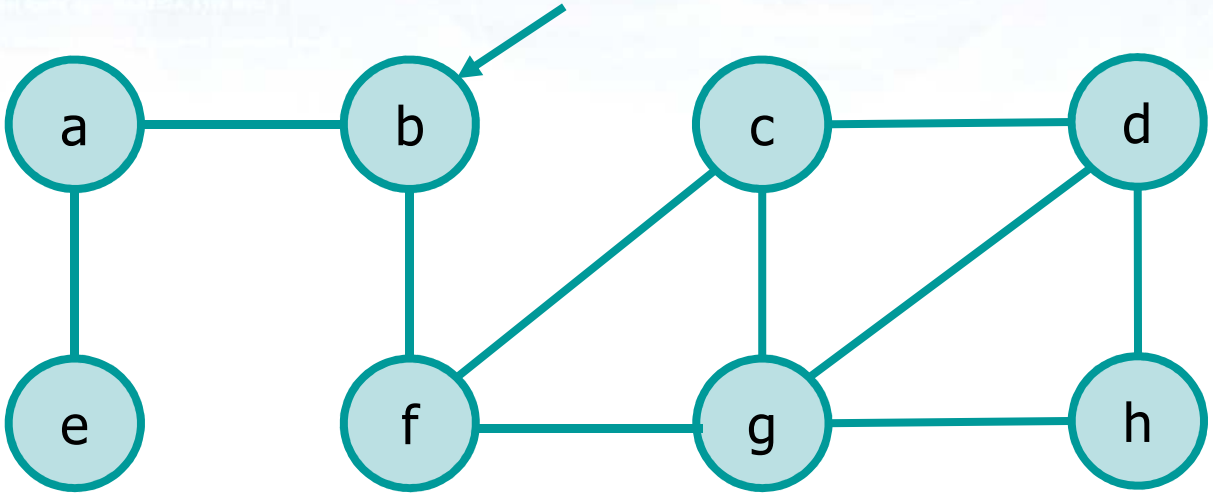
Breadth-first search

- Starting from a node s
  - It identifies all nodes reachable from the source node **s**
  - It computes the minimum distance from s to all the nodes reachable from s
  - It generates a BFS tree
- Breadth-first
  - It expands in parallel the whole border between already discovered nodes and not yet discovered nodes
  - It uses a FIFO queue

Breadth-first search

- Discovery of a node
  - The first time the node is encountered during the visit
- Nodes
  - White
    - Not yet discovered
  - Gray
    - Discovered but not yet completed
  - Black
    - Discovered and completed
- Given a node u
  - st[u]: parent of u in the BFS tree

Example

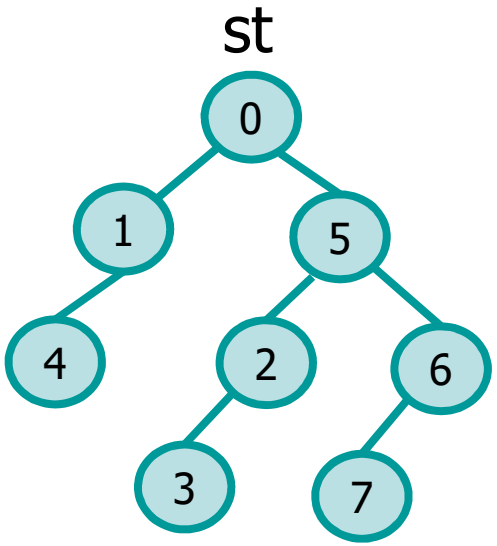


st

a	b	c	d	e	f	g	h
-1	-1	-1	-1	-1	-1	-1	-1

Property

- Shortest path
  - BFS finds the shorest path between s and all the nodes reachable from s
- Example
  - Shortest path from 0 to 3
  - 0, 5, 2, 3
  - length = 3



Graph library (with adjacency list)

```
g = graph_load(argv[1], &nv);
printf("Initial vertex? ");
scanf("%d", &i);
n = graph_find(g, nv, i);
graph_bfs(g, nv, n);

n = g;
printf("List of vertices:\n");
while (n != NULL) {
  if (n->color != WHITE) {
    printf("%2d: %d (%d)\n",
      n->id, n->dist, n->pred ? n->pred->id : -1);
  }
  n = n->next;
}
graph_dispose(g, nv);
```

Printing BFS info

Graph library (with adjacency list)

Initialization $\forall v \in V$ as WHITE, INT_MAX, NULL is performed during load

```
void graph_bfs(graph_t *g, int nv, graph_t *n) {
  queue_t *qp=queue_init(nv);
  graph_t *d;
  edge_t *e;

  n->color = GREY;
  n->dist = 0;
  n->pred = NULL;
  queue_put(qp, (void *)n);
```

Graph library (with adjacency list)

```
  while (!queue_empty_m(qp)) {
    queue_get(qp, (void **)&n);
    e = n->head;
    while (e != NULL) {
      d = e->dst;
      if (d->color == WHITE) {
        d->color = GREY;
        d->dist = n->dist + 1;
        d->pred = n;
        queue_put(qp, (void *)d);
      }
      e = e->next;
    }
    n->color = BLACK;
  }
  queue_dispose(qp, NULL);
}
```

Complexity

- Initialization $\forall v \in V$, O(V)
- Operations on the queue
  - The cost to enqueue and dequeue a vertex is O(1)
  - All vertices have to be inserted into the queue, i.e., the total cost is O(V)
- The procedure scans all adjacency lists
  - The sum of the length of all lists is $\Theta(E)$
  - The cost to manage them is O(E)
- Gobally the cost is
  - T(n) = O(|V|+|E|)

Depth-first search

- Given a connected (or unconnected) graph, starting from a source node **s**
  - It visits all the nodes of the graph (no matter they are reachable from s or not)
  - It labels each node v with its discovery time/endprocessing time pre[v]/post[v]
- It labels each edge
  - Directed graphs
    - T(ree), B(ackward), F(orward), C(ross)
  - Undirected graphs
    - T(ree), B(ackward)
- It generates a forest of DFS trees

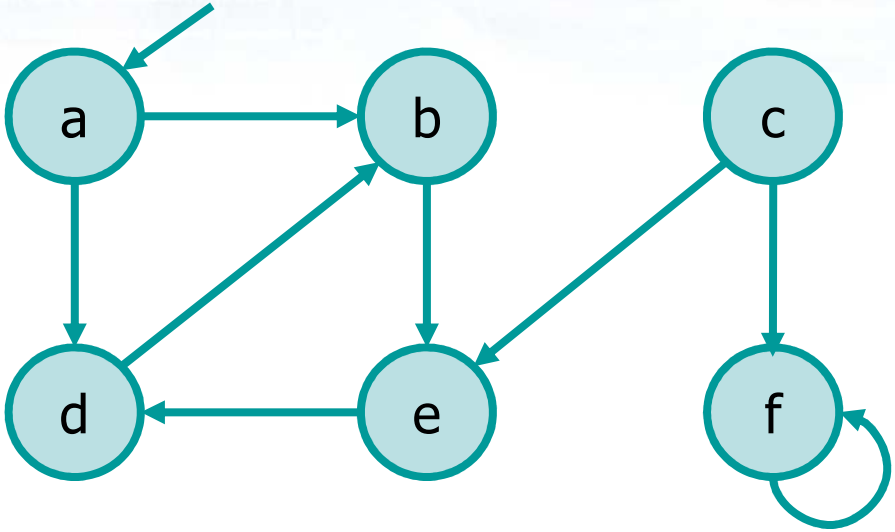
Principles

- Depth
  - It expands the last discovered node that has still undiscovered adjacent nodes
- Node discovery
  - First time the node is encountered in the visit (recursive descent, pre-order visit)
- Completion
  - End of node processing (exit from recursion, post-order visit)
- Discovery/Endprocessing
  - Discrete time that evolves according to counter time

Principles

- Nodes are conceptually classified as
  - White
    - Not yet discovered nodes
  - Gray
    - Already discovered, but not yet completed
  - Black
    - Discovered and completed
- For each node we store
  - Its discovery time pre[i]
  - Its endprocessing time post[i]
  - Its parent in the depth-first visit st[i]

Example

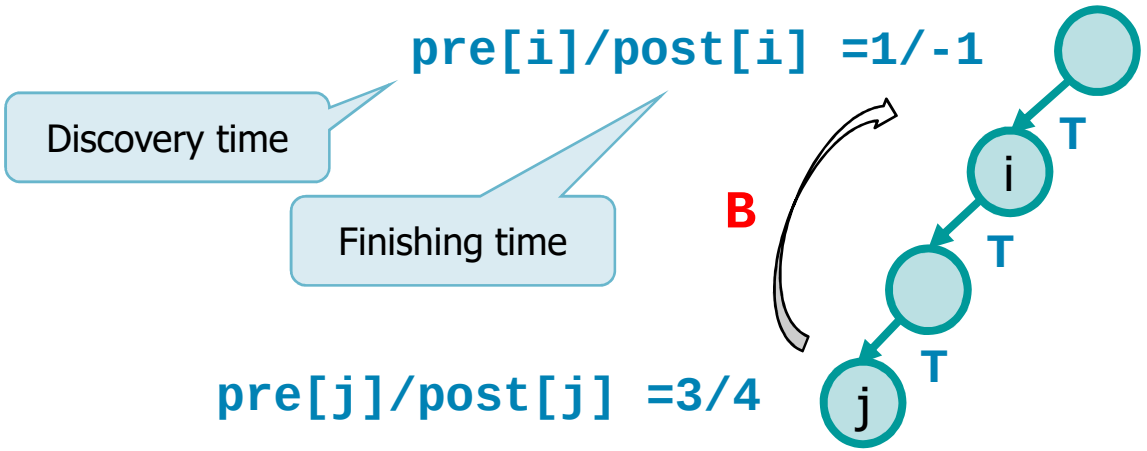


st

A	B	C	D	E	F
-1	-1	-1	-1	-1	-1

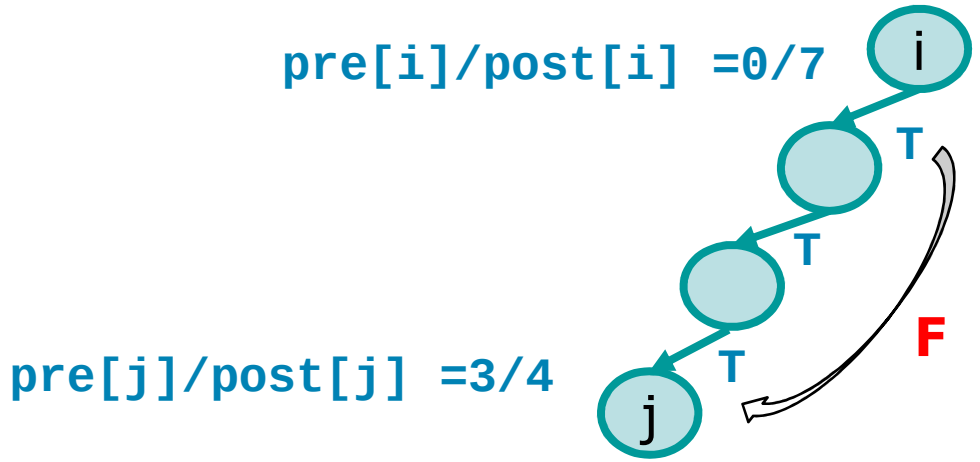
Edge labelling

- Directed graph
  - Tree (T)
    - Edges of the DFS tree
  - Back (B)
    - Connect a node j to an ancestor i in the tree
    - endprocessing time of i > endprocessing time of j
    - i has not yet become "black" post[i] == -1



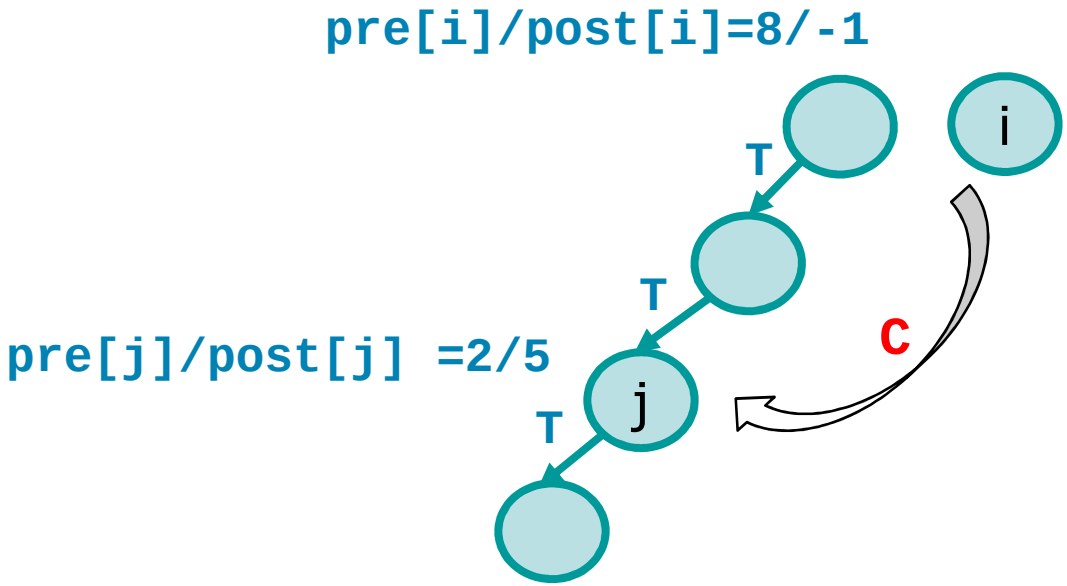
Edge labelling

- Forward (F)
  - Connect a node i to a descendant j in the tree
  - Discovery time of i < discovery time of j
    - pre[i] < pre[j]



Edge labelling

- Cross (C)
  - Remaining edges
  - Discovery time of i > discovery time of j
  - pre[i] > pre[j]



Edge labelling

- Undirected graph
  - Tree edges are defined as before
  - Backward edge are defined as before
  - Forward edges can be traversed both ways, then they are actually B edges
    - pre[j] > pre[i]
  - Cross edges can be visited both ways, then thay are Tree edges
    - pre[j] < pre[i]

Graph library (with adjacency list)

```
g = graph_load(argv[1], &nv);

printf("Initial vertex? ");
scanf("%d", &i);
graph_dfs(g, nv, i);

graph_dispose(g, nv);
```

DFS
(recursive function)

Initialization $\forall v \in V$ as WHITE, NULL is performed during load

Graph library (with adjacency list)

```
void graph_dfs(graph_t *g, int nv, int i) {
  int currTime=0;

  printf("List of edges:\n");
  currTime = graph_dfs_r(g, nv, i, currTime);
  for (i=0; i<nv; i++) {
    if (g[i].color == WHITE) {
      currTime = graph_dfs_r(g, nv, i, currTime);
    }
  }
  printf("List of vertices:\n");
  for (i=0; i<nv; i++) {
    printf("%2d: %2d/%2d (%d)\n",
      i, g[i].td, g[i].tq, g[i].pred);
  }
}
```

Graph library (with adjacency list)

```
int graph_dfs_r(
  graph_t *g, int nv, int i, int currTime) {
  edge_t *e;
  int j;
  g[i].color = GREY;
  g[i].td = ++currTime;
  e = g[i].head;
  while (e != NULL) {
    j = e->dst;
    switch (g[j].color) {
      case WHITE: printf("%d -> %d : T\n", i, j); break;
      case GREY : printf("%d -> %d : B\n", i, j); break;
      case BLACK: if (g[i].td < g[j].td) {
                     printf("%d -> %d : F\n", i, j);
                  } else {
                     printf("%d -> %d : C\n", i, j);
                  }
    }
```

Graph library (with adjacency list)

```
    if (g[j].color == WHITE) {
      g[j].pred = i;
      currTime = graph_dfs_r(g, nv, j, currTime);
    }
    e = e->next;
  }
  g[i].color = BLACK;
  g[i].tq = ++currTime;

  return currTime;
}
```

Complexity

- DFS is called once for each vertex $v \in V$
- During each call the adjacency list of v is visited
- Since the length of all adjacency list is $\Theta(|E|)$ the total cost is $\Theta(|E|)$
- The overall cost is $\Theta(|V|+|E|)$