# Graphs

Computing Lab

http://www.isical.ac.in/~dfslab

# Representation (adjacency list)

```
typedef struct {
    unsigned int degree;
    unsigned int *neighbours;
    float *edge_weights;
} GNODE;

typedef struct {
    unsigned int num_vertices, num_edges; // or  n, m  or  v, e  for
     short
    GNODE *adj_list;
    unsigned int num_components, *component_labels;
    PRINTER_FN print_vertex;
} GRAPH;

typedef struct {
    unsigned int length, *vertices;
} PATH;
```

# Utility functions

See `utils.c` for implementation details.

```
extern int read_graph(GRAPH *g, FILE *fp, bool weighted_flag);
extern void free_graph(GRAPH *g, bool weighted_flag);
extern void print_graph(GRAPH *g, bool weighted_flag);
```

# Traversals (traversals.c)

## Main functions

```
extern int dfs(GRAPH *g, unsigned int start, PATH *sequence);
extern int bfs(GRAPH *g, unsigned int start, PATH *sequence);
```

## Helper functions

```
extern int setup_traversal(GRAPH *g);
extern void cleanup_traversal(void);
```

- Traversals require auxiliary data structures.
- Size of these auxiliary data structures depends on size of graph.
- (Re)allocate memory if necessary, and **initialise** for *every new graph*.
- Free allocated memory at end.

# Digression: `previsit`, `postvisit` functions

- May need to do some additional work when a node is seen for the first time, and when it is fully processed.
- See `DPV` for more details.
- Use function pointers, `NULL` by default.

```
void (*previsit)(GRAPH *, unsigned int v);
void (*postvisit)(GRAPH *, unsigned int v);
```

# Code for DFS

```
if (previsit) previsit(g, start);
if (push(&stack, (void *) &start))
    return -1;
while (stack.top > 0) {
    assert(-1 != pop(&stack, (void *) &v));
    if (visited[v]) continue; // This check is needed. Why?
    visited[v] = true;
    if (postvisit) postvisit(g, v);
    if (sequence) sequence->vertices[sequence->length++] = v;
    else printf("Visited vertex %u\n", v); // Could go in postvisit()
    for (int i = g->adj_list[v].degree-1; i >= 0; i--)
        if (! visited[g->adj_list[v].neighbours[i]]) {
            if (previsit) previsit(g, g->adj_list[v].neighbours[i]);
            if (push(&stack, (void *) &(g->adj_list[v].neighbours[i])))
                return -1;
        }
}
```

# Connected components

```
static void assign_cc_label(GRAPH *g, uint v) {
    g->component_labels[v] = g->num_components;
    return;
}
```

---

```
previsit = assign_cc_label;
for (i = 0; i < g->num_vertices; i++)
    if (g->component_labels[i] == 0) {
        fprintf(stderr, "Labelling with %u from vertex %u\n", ++g->
num_components, i);
        if (dfs(g, i, NULL)) return -1;
    }
```

# Example graphs

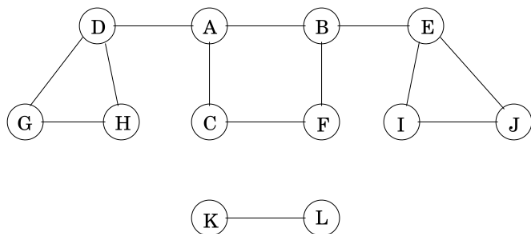



Figure 3.2 from ***DPV***

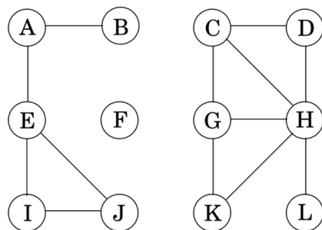



Figure 3.6 from ***DPV***

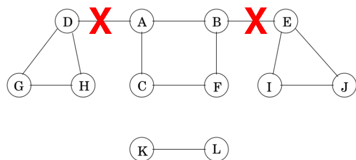



Figure 3.2 (modified) from ***DPV***

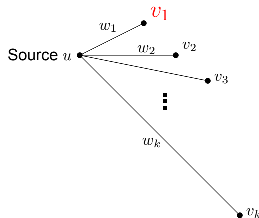
A dummy node (node 0, with degree 0) has been added to all the examples.

# Exercises I

0. Work out how to compile all the different `.c` files so that you can run `graph-driver.c`.
1. The provided implementations have not been thoroughly tested. Following the example inputs, create additional graphs and test the code. Please include pathological / corner cases in your tests, check for bugs and memory errors, and report.
2. Modify the type definitions in `graph.h` so that
   (a) each vertex can be assigned a name in the form of a string;
   (b) *each edge can be assigned a weight (positive or negative).*

   Think about what modifications would be necessary if you had to store a chunk of data of any type in each node.

# Shortest path algorithms: Dijkstra




$w_1 \leq w_2 \leq w_3 \leq \cdots \leq w_k$

**Q1.** Which vertex is the closest to $u$ in the entire graph?

## By induction

- One shortest path to each node in $V_{\text{explored}}$ has been found.
- Any unexplored node is farther away from $u$ than any explored node.
- Current distances correspond to shortest paths that go only through explored vertices.

# Dijkstra's algorithm

```
for all u ∈ V:
    dist(u) = ∞
    prev(u) = nil
dist(s) = 0

H = makequeue(V)   (using dist-values as keys)
while H is not empty:
    u = deletemin(H)
    for all edges (u,v) ∈ E:
        if dist(v) > dist(u) + l(u,v):
            dist(v) = dist(u) + l(u,v)
            prev(v) = u
            decreasekey(H,v)
```

$m \triangleq |E|$
$n \triangleq |V|$

This (makeheap / buildheap / heapify) takes $O(n)$ time.

Each *deletemin* takes $O(\lg n)$ time; it is called $O(n)$ times.

Need to add this to our heap library.
HOW?
How much time will it take?

# Shortest path algorithms: Bellman-Ford

```
static bool update_all_vertices(GRAPH *g, SHORTEST_PATH_INFO *p) {
    bool flag = false;
    unsigned int u, v, j;

    for (u = 0; u < g->num_vertices; u++) {
        for (j = 0; j < g->adj_list[u].degree; j++) {
            v = g->adj_list[u].neighbours[j];
            if (p[v].distance > p[u].distance + g->adj_list[u].
    edge_weights[j]) {
                p[v].distance = p[u].distance + g->adj_list[u].
    edge_weights[j];
                p[v].predecessor = u;
                flag = true;
            }
        }
    }
```

# Shortest path algorithms: Bellman-Ford

```
for (i = 0; i < g->num_vertices; i++) {
    updates_happened = update_all_vertices(g, p);
    if (!updates_happened) break;
}
if (i == g->num_vertices) return -1; // negative cycle exists
```

**Invariant**

After `i` iterations have been completed, shortest path consisting of at most `i` hops from `source` to all vertices have been found.

# Exercises

1. Test the new implementation of the (generic) heap by using it to sort a list of
   (a) integers,
   (b) floating point numbers, and
   (c) strings.
2. Create an example graph containing negative edge weights that demonstrates that Dijkstra's algorithm does not correctly work on such graphs.
3. Test the provided implementation on your counterexample, and report bugs.
4. Create an example graph containing a negative cycle. Print the minimum distances obtained after each iteration of the loop in the provided implementation of the Bellman Ford algorithm, and report bugs.

# Reference

- [DPV] *Algorithms* by Dasgupta, Papadimitriou, Vazirani.
  (Chapters 3, 4, 5)
- [CLRS] *Introduction to Algorithms* (2nd ed.) by Cormen, Leiserson, Rivest, Stein.
  (Section 6.5, pp. 139)