Lab Session 9

A*, Floyd–Warshall, Max Flow & Bipartite Matching

The Art of Computer Programming 2 — Week 9 — April 22, 2026

Objectives

By the end of this lab, you will have used graph algorithms to solve four realistic problems: route planning, social-network analysis, infrastructure provisioning, and resource allocation. More precisely, you should be able to:

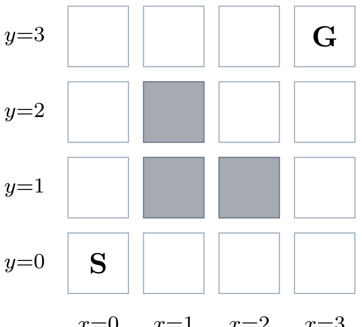
- Implement A* search with an admissible heuristic and trace it by hand
- Run Floyd–Warshall and use it to compute centrality measures on a graph
- Implement Ford–Fulkerson / Edmonds–Karp and detect a min cut
- Reduce bipartite matching to max flow and solve an assignment problem
- Choose the right algorithm for a given application

1 Warm-up by Hand (15 min)

Pen and paper — no computer needed.

1.1 A* on a small grid

You stand at $S = (0, 0)$ on a 4×4 grid and want to reach $G = (3, 3)$. You can move **up, down, left, right** (cost 1 per step). The grey cells are walls (impassable).



Recall

A* pops the open node with smallest $f(v) = g(v) + h(v)$, where g is the true cost from S and h is a heuristic estimate of the cost to G. The Manhattan heuristic $h(v) = |x_v - x_G| + |y_v - y_G|$ is admissible for 4-directional grids with unit costs.

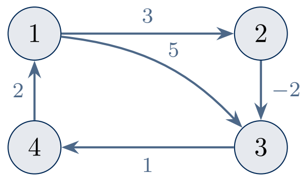
(a) Compute $h(v)$ (Manhattan distance to G) for all 13 free cells.

(b) Run A* from S until G is popped from the open set. For each iteration, give the popped node, its g, h, f values, and the successors added/updated. (Depending on how ties on f are broken, you will pop between 7 and 12 nodes before G.)

(c) How many cells does A* visit? How many would Dijkstra visit on the same grid? (Justify briefly.)

1.2 Floyd–Warshall on a tiny network

Apply Floyd–Warshall to the directed graph below to find all-pairs shortest paths.



(a) Write out the initial matrix $D^{(0)}$ (entries: 0 on diagonal, edge weight if edge exists, $+\infty$ otherwise).

(b) Compute $D^{(4)}$ (the final shortest-path matrix). Show the intermediate matrix $D^{(2)}$.

(c) Which pair of vertices has the longest shortest-path distance?

2 Route Planning with A* (20 min)

Real-world context

You are building the routing engine of a delivery app. Users enter a start address and a destination; you must return the shortest path on a city grid that includes one-way streets, closed roads, and pedestrian-only zones (which appear as walls). Dijkstra would explore in concentric circles around the start — wasteful when the destination is far away in a known direction. **A*** uses the geographical distance to the goal as a heuristic to focus the search.

2.1 Setting up the city grid

We model the city as a 2-D grid: each cell is either passable (cost 1 to enter) or a wall (impassable). Coordinates: (x, y) with $0 \leqslant x < W$, $0 \leqslant y < H$.

```
# Example city: 10x10 with a few buildings
GRID = [
    "..........",
    ".###..##..",
    ".#....##..",
    ".#.####...",
    "...#......",
    ".###.####.",
    "...#......",
    ".#.######.",
    ".#........",
    "..........",
]
# '.' = passable, '#' = wall
# Start = (0, 0) (top-left), Goal = (9, 9) (bottom-right)
```

2.2 Implement A*

```
import heapq

def manhattan(a, b):
    """Manhattan distance heuristic."""
```

```python
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def neighbors(grid, cell):
    """Yield passable 4-neighbours of cell = (x, y)."""
    x, y = cell
    H, W = len(grid), len(grid[0])
    for dx, dy in [(1,0), (-1,0), (0,1), (0,-1)]:
        nx, ny = x + dx, y + dy
        if 0 <= nx < W and 0 <= ny < H and grid[ny][nx] == '.':
            yield (nx, ny)

def astar(grid, start, goal, h=manhattan):
    """
    Returns (path, expanded) where:
      - path is the list of cells from start to goal (or None if unreachable)
      - expanded is the number of nodes popped from the open set
    """
    # YOUR CODE HERE
    # 1. Use a min-heap of (f, g, cell, parent)
    # 2. Track g_score[cell] = best g found so far
    # 3. Pop the node with smallest f. If it's the goal, reconstruct path.
    # 4. Otherwise expand its neighbours: tentative_g = g + 1
    #    If tentative_g < g_score.get(neighbour, +inf): update and push.
    # 5. Use a parent dict to reconstruct the path.
    pass
```

```python
# Test
path, expanded = astar(GRID, (0, 0), (9, 9))
print(f"Path length: {len(path) - 1}")   # number of edges
print(f"Cells expanded: {expanded}")
print(f"First 5 cells: {path[:5]}")
print(f"Last 5 cells:  {path[-5:]}")
assert path[0] == (0, 0) and path[-1] == (9, 9)
```

2.3 A* vs Dijkstra

```python
def dijkstra_grid(grid, start, goal):
    """Same signature as astar but uses h = 0 (i.e. Dijkstra)."""
    return astar(grid, start, goal, h=lambda a, b: 0)

p1, e1 = astar(GRID, (0, 0), (9, 9))
p2, e2 = dijkstra_grid(GRID, (0, 0), (9, 9))
print(f"A*       : path length {len(p1)-1}, {e1} cells expanded")
print(f"Dijkstra : path length {len(p2)-1}, {e2} cells expanded")
assert len(p1) == len(p2)  # both are optimal
```

Questions.

(a) Is the path *length* the same for A* and Dijkstra? Why?

(b) Compare the number of cells expanded. By what factor does A* win?

(c) What would happen with $h(v) = 2 \cdot \text{Manhattan}(v, \text{goal})$ (overestimating heuristic)? Try it: is the path still optimal?

Important

A non-admissible heuristic (one that overestimates) makes A* **lose optimality**. The algorithm will still terminate and return *some* path, but possibly not the shortest one. For Google Maps and friends, a careful admissible heuristic (Euclidean / great-circle distance) is non-negotiable.

3 Social Network Centrality with Floyd–Warshall (20 min)

Real-world context

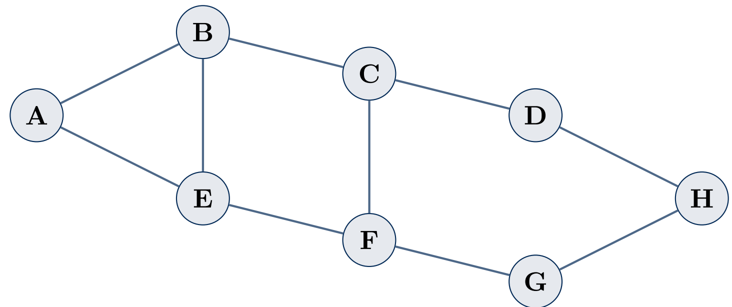
You are analysing a small online community to identify "influencers". Several centrality measures exist; two of the most popular are:

- **Closeness centrality:** a node is central if it is on average *close* to everyone else — $C(v) = (n-1)/\sum_{u \neq v} d(v,u)$.
- **Eccentricity:** the longest shortest path from v to any other node — the smaller it is, the more "in the middle" you are.

Both require **all-pairs shortest paths**, exactly what Floyd–Warshall computes in $\Theta(V^3)$.

3.1 The dataset

A friendship network of 8 people. An edge $u \leftrightarrow v$ means "u and v are friends" (undirected, weight 1).




```
NAMES = ["A", "B", "C", "D", "E", "F", "G", "H"]
N = 8
EDGES = [("A","B"), ("A","E"), ("B","C"), ("B","E"), ("C","D"),
         ("C","F"), ("E","F"), ("D","H"), ("F","G"), ("G","H")]
```

3.2 Implement Floyd–Warshall

```
INF = float("inf")

def floyd_warshall(n, weight):
    """
    Args:
        n: number of vertices
        weight: n x n matrix with weight[i][j] = edge weight (INF if no edge),
                weight[i][i] = 0.
    Returns:
        dist: n x n matrix of shortest-path distances.
    """
    # YOUR CODE HERE
    # dist = deep copy of weight
    # for k in range(n):
```

```python
    #     for i in range(n):
    #         for j in range(n):
    #             if dist[i][k] + dist[k][j] < dist[i][j]:
    #                 dist[i][j] = dist[i][k] + dist[k][j]
    pass

def build_weight_matrix(n, names, edges):
    """Build the symmetric weight matrix from an undirected edge list."""
    idx = {name: i for i, name in enumerate(names)}
    W = [[INF] * n for _ in range(n)]
    for i in range(n):
        W[i][i] = 0
    for u, v in edges:
        i, j = idx[u], idx[v]
        W[i][j] = W[j][i] = 1
    return W
```

3.3 Centrality measures

```python
def closeness(dist, i):
    """Closeness centrality of node i: (n-1) / sum of distances."""
    n = len(dist)
    s = sum(dist[i][j] for j in range(n) if j != i)
    return (n - 1) / s if s > 0 else 0.0

def eccentricity(dist, i):
    """Eccentricity of node i: max distance to any other node."""
    n = len(dist)
    return max(dist[i][j] for j in range(n) if j != i)

def diameter(dist):
    """Diameter: maximum eccentricity over all nodes."""
    return max(eccentricity(dist, i) for i in range(len(dist)))

# Run on the friendship graph
W = build_weight_matrix(N, NAMES, EDGES)
D = floyd_warshall(N, W)

print("Closeness centrality:")
ranking = sorted(range(N), key=lambda i: -closeness(D, i))
for i in ranking:
    print(f"  {NAMES[i]}: {closeness(D, i):.3f} (ecc={eccentricity(D, i)})")

print(f"\nDiameter of the network: {diameter(D)}")
print(f"Most central person: {NAMES[ranking[0]]}")
```

Questions.

(a) Who is the most central person? Does the answer match your visual intuition?

(b) What is the diameter? Identify a pair of friends realising it.

(c) Suppose you could add *one* new friendship edge to lower the diameter. Which edge would you add?

Hint

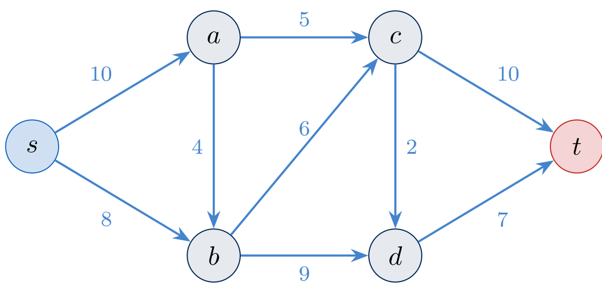
For sparse graphs ($E = \mathcal{O}(V)$), running Dijkstra from every vertex would take $\mathcal{O}(V \cdot E \log V) = \mathcal{O}(V^2 \log V)$, which beats Floyd–Warshall's $\mathcal{O}(V^3)$. But Floyd–Warshall is much simpler to implement, handles negative weights for free, and uses contiguous memory — ideal for the dense networks (a few thousand nodes) we deal with here.

4 Data-Center Bandwidth with Max Flow (25 min)

Real-world context

You operate a small data-centre with redundant fibre links between racks. A backup job must transfer as much data per second as possible from the **storage rack** (s) to the **archive rack** (t). Each link has a fixed bandwidth in Gb/s. What is the maximum throughput, and which links are the bottlenecks (so that you know which to upgrade)?

The network:



4.1 Implement Edmonds–Karp

Recall

Edmonds–Karp = Ford–Fulkerson with **BFS** for finding augmenting paths. This guarantees $\mathcal{O}(VE^2)$ runtime regardless of capacity values. We represent capacities by a dictionary `cap[(u,v)]`; the residual capacity of edge (u, v) is `cap[(u,v)] - flow[(u,v)] + flow[(v,u)]`.

```python
from collections import defaultdict, deque

def edmonds_karp(graph_caps, s, t):
    """
    Args:
        graph_caps: dict (u, v) -> capacity (only forward edges listed).
        s: source name; t: sink name.
    Returns:
        (max_flow, flow_dict) where flow_dict[(u, v)] is the flow on edge (u,v).
    """
    # Build adjacency: each edge (u, v) creates a residual reverse edge (v, u).
    nbrs = defaultdict(set)
    for (u, v) in graph_caps:
        nbrs[u].add(v)
        nbrs[v].add(u)
    cap = dict(graph_caps)           # original capacities
    flow = defaultdict(int)          # flow on every edge (incl. reverse)
```

```python
    def residual(u, v):
        # forward residual = capacity - flow on (u,v); plus flow we can cancel on (v,u)
        return cap.get((u, v), 0) - flow[(u, v)] + flow[(v, u)]

    def bfs_path():
        """Return a list [s, ..., t] of nodes forming an augmenting path, or None."""
        # YOUR CODE HERE
        # Standard BFS from s, only crossing edges with residual(u, v) > 0.
        pass

    max_flow = 0
    while True:
        path = bfs_path()
        if path is None:
            break
        # Bottleneck along the path
        bn = min(residual(path[i], path[i+1]) for i in range(len(path) - 1))
        # Push bn units
        for i in range(len(path) - 1):
            u, v = path[i], path[i+1]
            # Cancel reverse flow first, then add forward
            cancel = min(bn, flow[(v, u)])
            flow[(v, u)] -= cancel
            flow[(u, v)] += bn - cancel
        max_flow += bn

    return max_flow, dict(flow)
```

```python
caps = {
    ("s", "a"): 10, ("s", "b"): 8,
    ("a", "c"): 5,  ("a", "b"): 4,
    ("b", "d"): 9,  ("b", "c"): 6,
    ("c", "t"): 10, ("d", "t"): 7,
    ("c", "d"): 2,
}

mf, fl = edmonds_karp(caps, "s", "t")
print(f"Maximum flow s -> t : {mf} Gb/s")
for (u, v), f in sorted(fl.items()):
    if f > 0 and (u, v) in caps:
        print(f"  {u} -> {v} : {f} / {caps[(u, v)]}")
```

4.2 Find the bottleneck (min cut)

Recall

By Max-Flow Min-Cut, the bottleneck of the network is a **minimum cut**: the set S of nodes reachable from s in the *final residual graph*, and $T = V \setminus S$. The saturated edges from S to T are the ones to upgrade first.

```python
def min_cut(graph_caps, flow, s):
    """Return (S, T, cut_edges) from the residual graph after max flow."""
    # YOUR CODE HERE
    # 1. Build the residual graph from graph_caps and flow.
    # 2. BFS/DFS from s using only edges with positive residual.
    # 3. S = visited; T = everyone else; cut_edges = original edges (u,v)
```

```
    #    with u in S and v in T.
    pass

S, T, cut = min_cut(caps, fl, "s")
print(f"S = {sorted(S)}")
print(f"T = {sorted(T)}")
print(f"Cut edges (to upgrade): {cut}")
print(f"Sum of cut capacities: {sum(caps[e] for e in cut)}")
```

Questions.

(a) What is the maximum throughput of the network?

(b) Which physical links should you upgrade to increase the throughput?

(c) If you double the capacity of *one* cut edge, by how much does the max flow increase? Try it.

5 Intern–Project Assignment via Bipartite Matching (15 min)

Real-world context

Six interns are about to start the summer at PSL. Each intern has listed the projects they are qualified for, and the lab can host at most one intern per project. Your task as program director: assign as many interns as possible to a project they like.
This is the classic **bipartite matching** problem, which we solve by reduction to max flow: add a super-source s connected to each intern (capacity 1), a super-sink t connected from each project (capacity 1), and an edge of capacity 1 from intern i to project p whenever i likes p.

5.1 The data

```
INTERNS  = ["Alice", "Bob", "Carol", "David", "Eva", "Faith"]
PROJECTS = ["NLP", "RL", "Vision", "Robotics", "Theory"]

# preferences[i] = list of projects intern i is qualified for
PREFERENCES = {
    "Alice":  ["NLP", "Vision"],
    "Bob":    ["RL", "Robotics"],
    "Carol":  ["NLP", "Theory"],
    "David":  ["Vision", "Robotics"],
    "Eva":    ["NLP", "RL"],
    "Faith":  ["Theory"],
}
```

5.2 Reduction to max flow

```
def bipartite_matching(left, right, edges):
    """
    Args:
        left:  list of names on the left side
        right: list of names on the right side
        edges: list of (l, r) pairs denoting allowed matches
    Returns:
        list of (l, r) pairs forming a maximum matching.
    """
```

```python
    # YOUR CODE HERE
    # 1. Build capacities:
    #      ("s", l) = 1 for each l in left
    #      (l, r)   = 1 for each (l, r) in edges
    #      (r, "t") = 1 for each r in right
    # 2. Run edmonds_karp; the matching is the set of (l, r) edges with flow 1.
    pass

edges = [(i, p) for i, prefs in PREFERENCES.items() for p in prefs]
matching = bipartite_matching(INTERNS, PROJECTS, edges)

print(f"Maximum matching size: {len(matching)}")
for i, p in matching:
    print(f"  {i}  ->  {p}")
```

Questions.

(a) How many interns get a project? Are any left out?

(b) There are several maximum matchings of the same size. Why does the *size* not depend on the order in which Edmonds–Karp finds augmenting paths?

(c) Suppose Faith broadens her interests to also accept “Vision”. Does the matching size change?

Important

The reduction works because all capacities are 1: each intern-node carries at most one unit of flow (from s), and each project-node accepts at most one unit (to t). **Integrality** of Ford–Fulkerson on integer capacities then guarantees a 0/1 flow, i.e. a valid matching.

Bonus — Image Segmentation as Min Cut 🏆

Bonus

A classic computer-vision trick: separate *foreground* from *background* in a grey-scale image by computing a min cut on the pixel graph.

- Build a graph with one node per pixel plus a source s (“foreground”) and a sink t (“background”).
- Add edge $s \to p$ with capacity proportional to “how foreground-y pixel p looks” (e.g. how dark it is).
- Add edge $p \to t$ with capacity proportional to “how background-y pixel p looks” (e.g. how bright it is).
- Connect each pair of neighbouring pixels with an edge whose capacity discourages cuts at smooth gradients (large where pixels look similar).
- Run max flow. The min cut separates foreground from background pixels.

Mini-task. Take a 5×5 toy “image” (a Python matrix of integers 0–9), build the graph as above with capacities you choose, run your `edmonds_karp`, and visualise which pixels end up on the source side versus the sink side. *This is exactly how interactive segmentation works in tools like GrabCut.*