1 Warm-up by Hand

1.1 A* on a small grid

Solution

(a) Manhattan heuristic $h(v) = |x_v - 3| + |y_v - 3|$, computed for each free cell. Walls are at $(1,1), (2,1), (1,2)$; reading the grid with y increasing upwards:

$y = 3$	3	2	1	0
$y = 2$	4	#	2	1
$y = 1$	5	#	#	2
$y = 0$	6	5	4	3
	$x = 0$	$x = 1$	$x = 2$	$x = 3$

(b) The optimal path has length 6 (= Manhattan distance), so every cell on the optimal frontier has $f = g + h = 6$. A* pops nodes in roughly this order (ties broken arbitrarily by the heap):

#	pop	g	h	f	action
1	$(0,0)$	0	6	6	expand → push $(1,0), (0,1)$
2	$(1,0)$	1	5	6	push $(2,0)$
3	$(0,1)$	1	5	6	push $(0,2)$
4	$(2,0)$	2	4	6	push $(3,0)$
5	$(0,2)$	2	4	6	push $(0,3)$
6	$(3,0)$	3	3	6	push $(3,1)$
7	$(0,3)$	3	3	6	push $(1,3)$
8	$(3,1)$	4	2	6	push $(3,2)$
…followed by $(1,3), (3,2), (2,3), (3,3)$.					

A* pops $(3,3)$ at step 12 with $g = 6$. Path: $(0,0) \to (1,0) \to (2,0) \to (3,0) \to (3,1) \to (3,2) \to (3,3)$ (one of several minimum-length paths).

(c) A* visits about **12** of the 13 free cells. Dijkstra would visit all 13. The gain is small here because every free cell has $f \leqslant 6$ (the Manhattan heuristic is tight everywhere along the "triangle" between S and G). On larger open grids where most cells lie *behind* the source, the h value lifts their f above the optimum and A* skips them entirely — this is the "focused" search you will see in part §2.

Common Mistakes

- Forgetting to add 1 (the edge cost) to g when expanding.
- Computing f from a stale g when the heap holds two entries for the same cell — always recompute or use lazy deletion.
- Claiming A* is "always strictly faster" than Dijkstra. False: in the worst case (uninformative h, or all cells on the optimal frontier) they expand the same nodes.

1.2 Floyd–Warshall on a tiny network

Solution

(a) Initial matrix $D^{(0)}$ (rows/columns indexed $1, 2, 3, 4$):

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 5 & \infty \\ \infty & 0 & -2 & \infty \\ \infty & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{pmatrix}$$

Intermediate $D^{(2)}$ (after allowing intermediate vertex 1 then 2):

$$D^{(2)} = \begin{pmatrix} 0 & 3 & \mathbf{1} & \infty \\ \infty & 0 & -2 & \infty \\ \infty & \infty & 0 & 1 \\ 2 & \mathbf{5} & \mathbf{3} & 0 \end{pmatrix}$$

The bold entries are new: e.g. $D^{(2)}[1,3] = D^{(1)}[1,2] + D^{(1)}[2,3] = 3 + (-2) = 1$ improves on the direct edge of weight 5.

(b) Final matrix:

$$D^{(4)} = \begin{pmatrix} 0 & 3 & 1 & 2 \\ 1 & 0 & -2 & -1 \\ 3 & 6 & 0 & 1 \\ 2 & 5 & 3 & 0 \end{pmatrix}$$

(c) The longest shortest-path distance is $\delta(3,2) = 6$: the path is $3 \to 4 \to 1 \to 2$ of weight $1 + 2 + 3 = 6$.

Explanation

A useful sanity check: every vertex sits on a directed cycle ($1 \to 2 \to 3 \to 4 \to 1$ of weight 4), so all entries of $D^{(4)}$ are finite. There is no negative cycle: the diagonal stays 0.

2 Application 1 — Route Planning with A*

2.1 Implementation

Solution

```python
def astar(grid, start, goal, h=manhattan):
    open_h = [(h(start, goal), 0, start)]
    parent = {start: None}
    g_score = {start: 0}
    expanded = 0
    while open_h:
        f, g, u = heapq.heappop(open_h)
        if g > g_score[u]:        # stale entry: a better path was found
            continue
        expanded += 1
        if u == goal:
            # reconstruct path
            path = []
            cur = u
            while cur is not None:
                path.append(cur)
                cur = parent[cur]
            return list(reversed(path)), expanded
        for v in neighbors(grid, u):
            tentative = g + 1
            if tentative < g_score.get(v, float("inf")):
                g_score[v] = tentative
                parent[v] = u
                heapq.heappush(open_h, (tentative + h(v, goal), tentative, v))
    return None, expanded   # no path
```

2.2 A* vs Dijkstra

Solution

On the 10×10 city grid (which contains 70 free cells out of 100), both algorithms return paths of length **18** (several optimal paths exist). Typical expansion counts:

Algorithm	Cells expanded
A* (Manhattan)	≈ 56
Dijkstra ($h = 0$)	≈ 69 (essentially every reachable cell)

(a) Same path length — both are optimal. With an admissible heuristic, A* retains optimality: no shorter path exists, so both algorithms return the same minimum length.
(b) A* wins by a factor of roughly **1.2**× on this modest grid. The improvement looks small because the maze leaves few "useless" cells far behind the start: every reachable cell has $f \leqslant 18$, so the heuristic cannot prune much. On larger or more open maps, A* wins by orders of magnitude (city-scale street graphs).
(c) With $h(v) = 2 \cdot$ Manhattan, the heuristic **overestimates** the true distance; A* becomes a kind of greedy best-first search and may return a **suboptimal** path. Concretely, it will commit early to a path that "looks" close to the goal and refuse to backtrack, even when a globally shorter detour exists.

Explanation

When the heuristic is consistent (the strong form of admissibility), f-values along any path are non-decreasing, so once a node is popped its g-value is provably optimal — the same argument as Dijkstra. With inadmissible h, this invariant breaks: you may pop the goal too early, before the optimal path has been discovered.

Common Mistakes

Make sure the implementation correctly handles stale heap entries (`g > g_score[u]: continue`). A common bug is to mark a node "closed" on the first pop and never revisit — for consistent heuristics this is fine, but with inconsistent (only admissible) heuristics it causes incorrect answers. Mention this if asked.

3 Application 2 — Social Network Centrality

3.1 Implementation

Solution

```
def floyd_warshall(n, weight):
    dist = [row[:] for row in weight]
    for k in range(n):
        for i in range(n):
            dik = dist[i][k]
            if dik == INF:
                continue
            row_i = dist[i]
            row_k = dist[k]
            for j in range(n):
                alt = dik + row_k[j]
                if alt < row_i[j]:
                    row_i[j] = alt
    return dist
```

3.2 Centrality results

Solution

After running Floyd–Warshall:

Person	Closeness	Eccentricity
A	0.438	4
B	0.538	3
C	**0.636**	**2**
D	0.500	3
E	0.538	3
F	**0.636**	**2**
G	0.500	3
H	0.438	4

(a) C and F tie for most central, both with closeness 0.636 and eccentricity 2. This matches intuition: looking at the picture, they sit at the "waist" of the graph, bridging the left cluster $\{A, B, E\}$ and the right cluster $\{D, G, H\}$.
(b) Diameter $= 4$, attained by the pair $A \leftrightarrow H$ (shortest path goes $A \to B \to C \to D \to H$ or $A \to E \to F \to G \to H$).
(c) Adding the edge $A \leftrightarrow H$ would obviously cut their distance to 1 and reduce the diameter to 3. A more interesting choice that also achieves diameter 3: $A \leftrightarrow G$ (cuts $\delta(A, H)$ to 2 via $A \to G \to H$, and cuts the eccentricity of both endpoints).

Explanation

Closeness is exactly $(n-1)/\sum_j d(v,j)$. For C and F: $\sum = 2+1+1+2+1+2+2 = 11$, so closeness $= 7/11 \approx 0.636$. The graph is almost a ladder; central rungs (C, F) score best, while the endpoints (A, H) suffer from longer average distances.

3.3 When to prefer Floyd–Warshall over Dijkstra-from-each-source?

Solution

For all-pairs shortest paths on n vertices and m edges:

- **Dense graphs** ($m = \Theta(n^2)$): Dijkstra-from-each-source is $\mathcal{O}(n \cdot (n+m)\log n) = \mathcal{O}(n^3 \log n)$, slower than Floyd–Warshall's $\mathcal{O}(n^3)$ *and* far more cache-unfriendly. Floyd–Warshall wins.
- **Sparse graphs** ($m = \mathcal{O}(n)$): Dijkstra-from-each-source is $\mathcal{O}(nm \log n) = \mathcal{O}(n^2 \log n)$, which beats $\mathcal{O}(n^3)$.
- **Negative weights, no negative cycle**: Floyd–Warshall handles them natively; Dijkstra cannot.

4 Application 3 — Data-Centre Bandwidth with Max Flow

4.1 Implementation of `bfs_path`

Solution

```
def bfs_path():
    parent = {s: None}
    q = deque([s])
    while q:
        u = q.popleft()
        if u == t:
            break
        for v in nbrs[u]:
            if v not in parent and residual(u, v) > 0:
                parent[v] = u
                q.append(v)
    if t not in parent:
        return None
    path, cur = [], t
    while cur is not None:
        path.append(cur)
        cur = parent[cur]
    return list(reversed(path))
```

4.2 Min-cut implementation

Solution

```python
def min_cut(graph_caps, flow, s):
    nbrs = defaultdict(set)
    for (u, v) in graph_caps:
        nbrs[u].add(v); nbrs[v].add(u)
    def res(u, v):
        return graph_caps.get((u,v), 0) - flow.get((u,v), 0) + flow.get((v,u), 0)
    S = {s}
    q = deque([s])
    while q:
        u = q.popleft()
        for v in nbrs[u]:
            if v not in S and res(u, v) > 0:
                S.add(v); q.append(v)
    T = set(nbrs.keys()) - S
    cut = [(u, v) for (u, v) in graph_caps if u in S and v in T]
    return S, T, cut
```

4.3 Numerical results

Solution

Edmonds–Karp finds four augmenting paths and terminates with $\boxed{\text{maximum flow} = 17 \text{ Gb/s}}$:

#	Augmenting path	Bottleneck	$\lvert f\rvert$ after
1	$s \to a \to c \to t$	5	5
2	$s \to b \to d \to t$	7	12
3	$s \to b \to c \to t$	1	13
4	$s \to a \to b \to c \to t$	4	17

Final flow on each link:

$$s\to a{:}9/10,\ s\to b{:}8/8,\ a\to c{:}5/5,\ a\to b{:}4/4,\ b\to c{:}5/6,\ b\to d{:}7/9,\ c\to t{:}10/10,\ d\to t{:}7/7.$$

Min cut. BFS in the residual graph from s reaches $\{s, a\}$ only (the saturated links $a \to b$, $a \to c$, $s \to b$ block further expansion). Hence

$$S = \{s, a\}, \qquad T = \{b, c, d, t\}, \qquad \text{cut edges} = \{(s, b), (a, c), (a, b)\}, \qquad c(S, T) = 8 + 5 + 4 = 17.$$

This confirms Max-Flow Min-Cut: $|f^*| = c(S, T) = 17$.
Question (a). Maximum throughput is 17 Gb/s.
Question (b). The bottlenecks are the three cut edges $(s, b), (a, c), (a, b)$. Upgrading any of these is the only way to raise the maximum flow. (Adding capacity on $c \to t$ or $d \to t$ would be wasted because they are not on the min cut.)
Question (c). Doubling the capacity of (s, b) from 8 to 16 lifts the cut $c(\{s, a\}, T)$ to $16{+}5{+}4 = 25$, but a new min cut may now bind. Re-running Edmonds–Karp gives a max flow of **17** Gb/s — unchanged because new bottlenecks $(c, t) = 10$ and $(d, t) = 7$ together with saturated $(a, c) = 5$ form another cut $\{s, a, b\}|\{c, d, t\}$ of capacity $5{+}6{+}9 = 20$, and the cut $\{s, a, b, c, d\}|\{t\}$ has capacity $10 + 7 = 17$. So the bottleneck just shifts to the sink-side cut once $(s, b), (a, c), (a, b)$ are relaxed. Doubling (c, t) *and* adding capacity on $(s, b), (a, c), (a, b)$ would actually move the needle.

Common Mistakes

- Treating the residual graph as if reverse edges did not exist — augmenting path 4 ($s \to a \to b \to c \to t$) uses the forward edge $a \to b$ but only succeeds because of the cancellation logic.

- Assuming the bottleneck of the network is simply the smallest individual edge capacity. The min cut is a *set* of edges, not a single one.
- Forgetting that BFS (Edmonds–Karp) is what guarantees polynomial runtime; DFS-based Ford–Fulkerson can take exponentially many iterations on adversarial inputs (the Zwick example from the slides).

5 Application 4 — Intern–Project Assignment

5.1 Implementation

Solution

```python
def bipartite_matching(left, right, edges):
    caps = {}
    for l in left:  caps[("s", l)] = 1
    for r in right: caps[(r, "t")] = 1
    for (l, r) in edges:
        caps[(l, r)] = 1
    _, flow = edmonds_karp(caps, "s", "t")
    return [(l, r) for (l, r) in edges if flow.get((l, r), 0) == 1]
```

5.2 Results

Solution

Maximum matching size = **5**. One realisation:

Intern	→	Project
Bob	→	Robotics
Carol	→	NLP
David	→	Vision
Eva	→	RL
Faith	→	Theory
Alice	—	*unmatched*

All five projects are filled; Alice is the one intern left out. Different runs of Edmonds–Karp may yield different assignments (e.g. Alice → NLP and Carol unmatched), but the size is always 5.
(a) 5 of the 6 interns get a project; Alice is left out.
(b) The size of a maximum matching is a property of the bipartite graph, not of the algorithm. Different choices of augmenting paths reach *different maximum matchings*, all of the same maximum size — this is the standard "many maxima, one optimum value" situation in combinatorial optimisation.
(c) If Faith additionally accepts "Vision", the matching size **remains 5**: there are still only 5 projects, and the previous matching already saturated all of them. Adding more options for Faith cannot increase the matching beyond $\min(|L|, |R|) = 5$.

Explanation

Why must Alice (or one specific intern) be left out? Because the maximum matching is bounded by both sides: $|M| \leqslant \min(|L|, |R|) = 5$. With 6 interns and 5 projects, at least one intern is

unmatched. The fact that *some* matching of size 5 exists is what we needed to verify.

Grading Notes

Accept any matching of size 5 with valid intern–project pairs. Students who hard-code the assignment without using the max-flow reduction lose half the marks: the point is to see the reduction work.

Bonus — Image Segmentation

Solution

A working sketch on a 5×5 "image" (values in 0–9):

```python
import math
img = [
    [9, 9, 9, 9, 9],
    [9, 0, 0, 0, 9],
    [9, 0, 0, 0, 9],
    [9, 0, 0, 0, 9],
    [9, 9, 9, 9, 9],
]
H, W = len(img), len(img[0])
caps = {}
LAMBDA = 5  # smoothness weight

# unary terms: dark pixels look "foreground", bright pixels look "background"
for y in range(H):
    for x in range(W):
        p = (x, y)
        caps[("s", p)] = 9 - img[y][x]   # darker => more foreground
        caps[(p, "t")] = img[y][x]       # brighter => more background

# pairwise smoothness
def gaussian(d, sigma=3.0):
    return math.exp(-(d * d) / (2 * sigma * sigma))

for y in range(H):
    for x in range(W):
        for dx, dy in [(1, 0), (0, 1)]:
            nx, ny = x + dx, y + dy
            if 0 <= nx < W and 0 <= ny < H:
                w = LAMBDA * gaussian(img[y][x] - img[ny][nx])
                caps[((x, y), (nx, ny))] = w
                caps[((nx, ny), (x, y))] = w

mf, flow = edmonds_karp(caps, "s", "t")
S, T, cut = min_cut(caps, flow, "s")
fg = sorted(p for p in S if isinstance(p, tuple))
print("Foreground pixels:", fg)
```

Expected output. The 9 dark pixels in the centre form the foreground; the 16 bright border pixels are background. The smoothness term prevents isolated pixels from defecting.

Explanation

This is the foundation of the famous **GrabCut** algorithm (Microsoft Research, SIGGRAPH 2004). Real implementations use richer unary terms (Gaussian Mixture Models in colour space)

and 8-neighbour pairwise terms, plus iterative re-fitting — but the algorithmic core is exactly what you implemented here: max-flow on a pixel graph.