Q1. A*-Algorithm

(a) Running Dijkstra’s algorithm using a binary heap (priority queue) on a graph $G=(V, E)$, there will be at most $|V|$ dequeue operations and $|V| + |E|$ enqueue operations, where both enqueue and dequeue takes $O(\log|V|)$, so the total complexity is $O((|V|+|E|)\log|V|)$. For the given grid there are $\sim N^2$ vertices and $\sim N^2$ edges, so the runtime is $O(N^2\log N)$.

(b) Dijkstra’s algorithm will find shortest from $s$ to all vertices, however it can be modified so it stops immediately when $t$ is explored. More specifically, for each iteration of updates, it checks if the current vertex dequeued is $t$; and if it is, return immediately. This way we explore all vertices that are at least as close to $s$ as $t$ is.

The exploration space is a diamond shape (and a perfect square, the yellow area above) centered on $s$ with radios $|m| + |n|$, where $\sim 2(|m| + |n|)^2$ vertices and edges are visited, hence the runtime $O((|m| + |n|)^2\log(|m|+|n|))$.

(c) $l'$ is never negative so Dijkstra’s algorithm can apply. Assuming the shortest path defined by $l'$ is found as: $s, r_1, r_2, ..., r_n, t$, then the shortest distance $\text{dist}'$ calculated is:

$$\text{dist}'(s, t) = l'(s, r_1) + l'(r_1, r_2) + ... + l'(r_n, t)$$

$$= l(s, r_1) - h(s) + h(r_1) + l(r_1, r_2) - h(r_1) + h(r_2) + ... + l(r_n, t) - h(r_n) + h(t)$$

$$= l(s, r_1) + l(r_1, r_2) + ... + l(r_n, t) + (h(t) - h(s))$$

$$= \text{dist}(s, t) + (h(t) - h(s))$$
Notice that $h(t) - h(s)$ is constant when both $t$ and $s$ are given, therefore the length $dist'$ of the path defined by $l'$ is the shortest if and only if the length $dist$ of same path defined by $l$ is the shortest.

(d) Given function $h(u) = |x_u - x_t| + |y_u - y_t|$, we have:
\[
l'(u, v) = l(u, v) - h(u) + h(v) = l(u, v) - (|x_u - x_t| + |y_u - y_t|) + (|x_v - x_t| + |y_v - y_t|) = 1 + ((|x_v - x_t| - |x_u - x_t|) + (|y_v - y_t| - |y_u - y_t|))\]

Notice that $l(u, v) = 1$, and this also means that either $x_u = x_v$ and $|y_u - y_v| = 1$, or $y_u = y_v$ and $|x_u - x_v| = 1$, therefore $l'(u, v) \geq 0$ and it satisfies the property in (c).

(e) First notice that the exploration space using the new modified Dijkstra's algorithm is the rectangle area (in yellow below). This is because for each vertex $w$ on the border of the rectangle, rather than choosing as the next step its neighbor vertex $v$ outside the rectangle, there must be another vertex $p$ (or $q$) inside to choose, such that $h(p) < h(v)$ and $l'(w, p) < l'(w, v)$.

Therefore, the total vertices and edges are $\sim |n|m|$, and in case $n$ or $m$ is zero, no more than $(|n| + 1)(|m| + 1)$, the runtime complexity for the modified algorithm is $O((|n| + 1)(|m| + 1)\log((|n| + 1)(|m| + 1)))$, which is asymptotically identical to $O((|n| + 1)(|m| + 1)\log(|n|m| + 2))$.

Q2. Tramp Steamer Problem

(a) When there is a negative cycle $C$: 
For some $C$, $\sum_{(i,j) \in C} w_{ij} = \sum_{(i,j) \in C} (rc_{ij} - p_j) = r \sum_{(i,j) \in C} c_{ij} - \sum_{(i,j) \in C} p_j < 0$

From this we know $r < (\sum_{(i,j) \in C} c_{ij})/(\sum_{(i,j) \in C} p_j)$. Since $r^*$ is defined as the maximum ratio, then we can conclude that $r < (\sum_{(i,j) \in C} c_{ij})/(\sum_{(i,j) \in C} p_j) \leq r^*$.

(b) Similarly, if every cycle has positive weight, then $\sum_{(i,j) \in C} w_{ij} = \sum_{(i,j) \in C} (rc_{ij} - p_j) = r \sum_{(i,j) \in C} c_{ij} - \sum_{(i,j) \in C} p_j > 0$ Therefore $r > (\sum_{(i,j) \in C} c_{ij})/(\sum_{(i,j) \in C} p_j)$ for all possible cycles in the graph. $r^*$ is defined as the maximum ratio of a cycle which must be one of the possible cycles, so it must hold that $r > r^*$.

(c) First Set 0 as the lower bound of $r^*$, and set $R = \max_{(i,j) \in E}(p_{ij}/c_{ij})$ to be the upper bound. Then perform binary search in this range to get more accurate estimation of $r^*$: for each round update each edge and set its weight $w_{ij} = rc_{ij} - p_j$, and run Bellman-Ford algorithm to detect negative cycles; if there is any ($r < r^*$), increase $r$; otherwise there are no negative cycles ($r \geq r^*$), decrease $r$. The search terminates when either (1) for the most recent two $r$-values ($r_{n-1}$ and $r_n$), $r_{n-1} > r^*$ and $r_n < r^*$ and $r_{n-1} - r_n < \varepsilon$, or (2) when there is a cycle whose weight is 0. Return the cycle with the weight $r_n$ or 0 as output.

$R$ is the maximum profit-to-cost ratio for any edge, so no edges have higher ratios and it is a secure upper bound for $r^*$. It is also trivial that 0 is a secure lower bound for $r^*$. As proven in (a) and (b), this algorithm will generate a sequence of $r$-values $r_1, r_2, \ldots r_n$ that converge to $r^*$.

The binary search is bounded by $O(\log(R/\varepsilon))$ iterations. Each iteration we perform Bellman-Ford algorithm that takes $O(|V||E|) = O(|V|^3)$. So the total runtime is $O(|V|^3 \log(R/\varepsilon))$.

Q3. Dijkstra and Shortest Paths

(a) By definition, each cycle of the graph is contained in one single strongly connected component. Since none of the edges within each component have negative weights, the cycle must have non-negative weight.

(b) For all of the edges into $C$, update the shortest distance of the target vertices (the “entry” vertices of $C$) from all components having an edge into $C$. Set shortest distances of all other vertices in $C$ as $+\infty$, and put all vertices in $C$ into a priority queue (a min heap, each vertex’s priority is the shortest distance value). Then within $C$, run Dijkstra’s algorithm until the shortest distances for all vertices are updated.

In this modified Dijkstra’s algorithm, the $dist(v)$ maintains the length of shortest path to $v$, where all edges but one ($e$) along the path are between vertices whose distances are already known. In each of the later steps we only remove one vertex with the smallest $dist(v)$ from the queue. Since any
path to \( v \) must contain \( e \) and other non-negative weighted edges, whenever \( v \) is removed from the queue its distance is the correct shortest length.

Setting initial shortest distances of all vertices of \( C \) will take at most \( |E| \) steps. Note that each edge is considered once so after setting all edges into \( C \), running Dijkstra’s algorithm within \( C \) will take extra \( O(|V| + |E|) \log|V|) \) time, so the total runtime is nearly linear.

(c) First find all strongly connected components (SCCs) of \( G \) and create its meta graph \( G' \), which is a DAG of all its SCCs. Then get the topological order of \( G' \). Within the SCC where \( s \) resides, run Dijkstra’s algorithm to get shortest distances for all of its vertices. Then run (b)’s algorithm for all other SCCs in their topological order. (For the case when there are no edges into the SCC from any of the previous SCCs, set all longest distances of its vertices \( +\infty \); in this case, there are not paths from \( s \) to any of its vertices.)

It takes linear time to create \( G' \) and get its topological order. Then running (b)’s algorithm for each SCC takes nearly linear time as proven in (b). The total runtime is therefore nearly linear.