1. Consider the following solitaire game, played on a connected undirected graph G. Initially, tokens are placed on three start vertices a, b, c. In each turn, you *must* move *all three* tokens, by moving each token along an edge from its current vertex to an adjacent vertex. At the end of each turn, the three tokens *must* lie on three different vertices. Your goal is to move the tokens onto three goal vertices x, y, z; it does not matter which token ends up on which goal vertex.

   Describe and analyze an algorithm to determine whether this puzzle is solvable. Your input consists of the graph G, the start vertices a, b, c, and the goal vertices x, y, z. Your output is a single bit: TRUE or FALSE. *[Hint: You've seen this sort of thing before.]*

Solution (product construction): We reduce to a reachability problem in a new graph $G' = (V', E')$ defined as follows:

- $V' = \{\{u, v, w\} \mid u, v, w$ are distinct vertices in $V\}$. Each vertex in V' is a ***set*** of exactly three vertices in V, indicating possible positions for the three tokens. There are $O(V^3)$ vertices in V'.
- $E' = \Big\{\big(\{r, s, t\}, \{u, v, w\}\big) \,\Big|\, ru, sv, tw \in E\Big\}$. Each edge in E' corresponds to a possible move by the three tokens. There are $O(E^3)$ edges in E'.
- The puzzle is solvable if and only if there is a path in G' from $\{a, b, c\}$ to $\{x, y, z\}$.
- Once we construct G', we can check for such a path by running whatever-first search from $\{a, b, c\}$.

The resulting algorithm runs in $O(V' + E') = O(V^3 + E^3) = \boldsymbol{O(E^3)}$ ***time***. ■

Rubric: 10 points, standard graph-reduction rubric. This is not the only correct solution.

2. Describe and analyze an algorithm to solve arbitrary acute-angle mazes. You are given a connected undirected graph G, whose vertices are points in the plane and whose edges are line segments; you are also given two vertices Start and Finish. Your algorithm should return TRUE if G contains a walk from Start to Finish that has only acute angles, and FALSE otherwise. Assume you have a subroutine that can determine in $O(1)$ time whether two segments with a common vertex define a straight, obtuse, right, or acute angle.

Solution: Let $G = (V, E)$ be the input graph. Imagine moving a token along a valid walk through G. At any time during the walk, the current ***state*** of the token is described by its current vertex and its previous vertex (if any). More formally, we reduce the angle-maze problem to a reachability problem in a new directed graph $G' = (V', E')$ as follows.

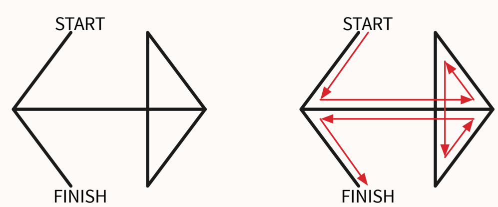
- $V' = \{(u, v) \mid uv \in E\} \cup \{\text{Start}\}$ — Each vertex (u, v) indicates that the token is currently at vertex v and it was previously at vertex u. These are *ordered* pairs; G' has *two* distinct vertices for each edge in G. In addition, we retain the Start vertex from G, because the token initially has no "previous" location.

  There are $2E + 1 = O(E)$ vertices altogether.
- There are two types of edges:
  - Starting edges $\{\text{Start}\to(\text{Start}, v) \mid (\text{Start}, v) \in E\}$ — At the beginning, the token can follow any edge out of the Start vertex.
  - Regular edges $\{(u, v)\to(v, w) \mid \angle uvw \text{ is straight or acute}\}$ — At every vertex in the walk after Start, the token can either continue straight or make an acute-angle turn.

  Edges are directed. The total number of edges is $O(EV)$, because there is at most one edge in G' for each $uv \in E$ and each $w \in V$.
- The maze is solvable if and only if the Start vertex can reach any vertex of the form (v, Finish) in G'.
- We can solve this problem using whatever-first search in G'. Specifically, we mark every vertex in G' that is reachable from Start, and then scan through all vertices (v, Finish) to see if any is marked.

Our overall algorithm runs in $O(V' + E') = \boldsymbol{O(EV)}$ ***time***. ■

Non-solution: It is not possible to solve the problem by running any version of WFS on the original input graph, because the solution walk can revisit the same vertex *up to four* times, but WFS always computes *simple paths*. See the maze below and its unique solution.



♣

Rubric: 10 points: standard graph-reduction rubric from HW6

- 2 for vertices
- 2 for edges (−1 for forgetting "directed")
- 2 for correct problem (reachability)
- 2 for correct algorithms (whatever-first search)
  - ½ for "depth" or "breadth" instead of "whatever"
  - 1 for Dijkstra
- 2 for time analysis (−1 for "$O(E^2)$")

3. Describe and analyze an algorithm to compute the length of the shortest Rectangle Walk in a given bitmap. Your input is an array $M[1..n, 1..n]$, where $M[i,j]=1$ indicates a black square and $M[i,j]=0$ indicates a white square. You can assume that a valid rectangle walk exists; in particular, $M[1,1]=0$ and $M[n,n]=0$.

Solution (two steps at a time, 10/10): We reduce to a shortest-path problem in a new undirected graph $G'=(V',E')$ as follows:

- $V'=\{(i,j) \mid M[i,j]=0\}$ is the set of all white pixels in M. Three are trivially at most $O(n^2)$ vertices in V'.
- $E'=\{(i^-,j^-)(i^+,j^+) \mid M[i^-,j^-]$ and $M[i^+,j^+]$ are opposite corners of a white rectangle in $M\}$. Each edge represents *two* steps in the Rectangle Walk: one that expands the rectangle from a single pixel at (i^-,j^-) to a rectangle containing both pixels, and one that contracts the rectangle to a single pixel at (i^+,j^+). Edges are undirected. There are trivially at most $O(n^4)$ edges in E'.
- We need to compute the shortest path from $(1,1)$ to (n,n) in G'. The length of this shortest path is *exactly half* the length of the shortest Rectangle Walk.
- Once we have computed G', we can compute this shortest path using breadth-first search in $O(V'+E') = \boldsymbol{O(n^4)}$ ***time***.

Unfortunately, we're not done, because brute-force construction of G' would take $O(n^6)$ time—for each of the $O(n^4)$ possible edges, we need $O(n^2)$ time to figure out whether the corresponding rectangle in M is all white. Fortunately, we can speed this up using dynamic programming.

For any indices $t \le b$ and $l \le r$, let $\textsc{Empty}(t,b,l,r) = \textsc{True}$ if every pixel in the subarray $M[t..b, l..r]$ is white, and $\textsc{False}$ otherwise. This function satisfies the following recurrence:

$$\textsc{Empty}(t,b,l,r) = \begin{cases} (M[t,l]=0) & \text{if } t=b \text{ and } l=r \\ (M[t,l]=0) \wedge \textsc{Empty}(t,t,l+1,r) & \text{if } t=b \text{ and } l<r \\ \textsc{Empty}(t,t,l,r) \wedge \textsc{Empty}(t+1,b,l,r) & \text{otherwise} \end{cases}$$

We can memoize this function into a four-dimensional array $\textsc{Empty}[1..n, 1..n, 1..n, 1..n]$. We can fill the array in $\boldsymbol{O(n^4)}$ ***time*** by considering t and l in decreasing order in the outer loops, and b and r in arbitrary order in the inner loops.

Once this array is filled, we can decide in $O(1)$ time whether a given pair of white pixels defines an edge in E', which means we can construct G' in $O(n^4)$ time. (Alternatively, instead of constructing an explicit adjacency list representation for G', we can pretend that G' is already represented as an adjacency matrix, in which case breadth-first search still runs in $O(V^2) = O(n^4)$ time.) We conclude that the entire algorithm runs in $\boldsymbol{O(n^4)}$ ***time***. ■

Solution (one step at a time, 8/10): We reduce to a shortest-path problem in a new undirected graph $G'=(V',E')$ as follows.

- G' has two types of vertices: individual white pixels in M, and maximal white rectangles in M. (A white rectangle is *maximal* if it does not fit inside a larger white rectangle.)

  – There are trivially at most $O(n^2)$ white pixels.
  – If we specify the indices of the top row (t), left column (l), and right column (r) of a maximal white rectangle, the index b of the bottom row is completely determined. (Specifically, b is the largest integer such that $b \le n$ and the subarray $M[t..b, l..r]$ is all white.) Thus, there are at most $O(n^3)$ maximal white rectangles.

- G' has an undirected edge between every white pixel $M[i,j]$ and every maximal white rectangle $M[t..b, l..r]$ such that $t \le i \le b$ and $l \le j \le r$. There are trivially at most $O(n^5)$ edges.
- Every Rectangle Walk in M corresponds with a unique walk from $M[1,1]$ to $M[n,n]$ in G. Thus, we need to compute the *shortest path* from $(1,1)$ to (n,n) in G'.
- Once we have computed G', we can compute this shortest path using breadth-first search in $O(V' + E') = \boldsymbol{O(n^5)}$ ***time***.

Unfortunately, we're not done, because brute-force construction of G' would take $O(n^6)$ time. For each of the $O(n^4)$ possible tuples (t, b, l, r), we need $O(n^2)$ time to determine whether the corresponding rectangle in M is a maximal white rectangle. Fortunately, we can speed this up using dynamic programming.

For any indices $t \le b$ and $l \le r$, let $\textsc{Empty}(t, b, l, r) = \textsc{True}$ if every pixel in the subarray $M[t..b, l..r]$ is white, and $\textsc{False}$ otherwise. This function satisfies the following recurrence:

$$\textsc{Empty}(t,b,l,r) = \begin{cases} (M[t,l]=0) & \text{if } t=b \text{ and } l=r \\ (M[t,l]=0) \wedge \textsc{Empty}(t,t,l+1,r) & \text{if } t=b \text{ and } l<r \\ \textsc{Empty}(t,t,l,r) \wedge \textsc{Empty}(t+1,b,l,r) & \text{otherwise} \end{cases}$$

We can memoize this function into a four-dimensional array $\textsc{Empty}[1..n, 1..n, 1..n, 1..n]$. We can fill the array in $\boldsymbol{O(n^4)}$ ***time*** by considering t and l in decreasing order in the outer loops, and b and r in arbitrary order in the inner loops.

Finally, the subarray $M[t..b, l..r]$ is a maximal white rectangle if and only if

$$\begin{aligned} \textsc{Empty}(t,b,l,r) \wedge \neg\textsc{Empty}(t-1,b,l,r) \wedge \neg\textsc{Empty}(t,b+1,l,r) \\ \wedge \neg\textsc{Empty}(t,b,l-1,r) \wedge \neg\textsc{Empty}(t,b,l,r+1) \end{aligned}$$

Thus, once the $\textsc{Empty}$ array is filled, we can decide in $O(1)$ time whether a given tuple (t, b, l, r) defines an vertex in V'. Once we have a list of all maximal white rectangles, we can enumerate the edges of G' in $O(n^5)$ time using two nested loops for each rectangle vertex. Thus, we can construct G' in $O(n^5)$ time, which implies that our entire algorithm runs in $\boldsymbol{O(n^5)}$ ***time***. ■

Rubric: 10 points = 5 for the reduction itself (graph modeling rubric) + 5 for the graph construction (dynamic programming rubric).

- −½ for “Dijkstra” instead of “breadth-first search”
- **Deadly Sins from the dynamic programming part apply to the entire problem.**
- Partial credit for slower algorithms: max 8 points for $O(n^5)$, 6 for $O(n^6)$, 5 for $O(n^7)$, 4 for $O(n^8)$; 3 for slower but correct. Scale partial credit.
- For slower algorithms that construct the graph by brute force, all points are from the graph reduction rubric (scaled to the maximum point values).
- If no details of the graph construction are given, ***assume*** that the construction is by brute force, and grade accordingly. In particular:
  - Using this exact graph reduction, without details of the graph construction, and reporting the running time as $O(n^6)$ is worth 6 points. $O(n^6)$ is the correct running time for brute-force construction.
  - Using this exact graph reduction, without details of the graph construction, and reporting the running time as $O(n^4)$ is worth 5 points, because the reported running time is incorrect.