Solution

Approach 1: Backtracking

**Overview**

If a hint is ever given on the problem description, that would be ***backtracking***.

Indeed, since the problem concerns about the *path exploration* in a *graph* data structure, it is a perfect scenario to apply the backtracking algorithm.

As a reminder, [backtracking](https://en.wikipedia.org/wiki/Backtracking) is a general algorithm that incrementally builds candidates to the solutions, and abandons a candidate (*"backtrack"*) as soon as it determines that the candidate cannot possibly lead to a valid solution.

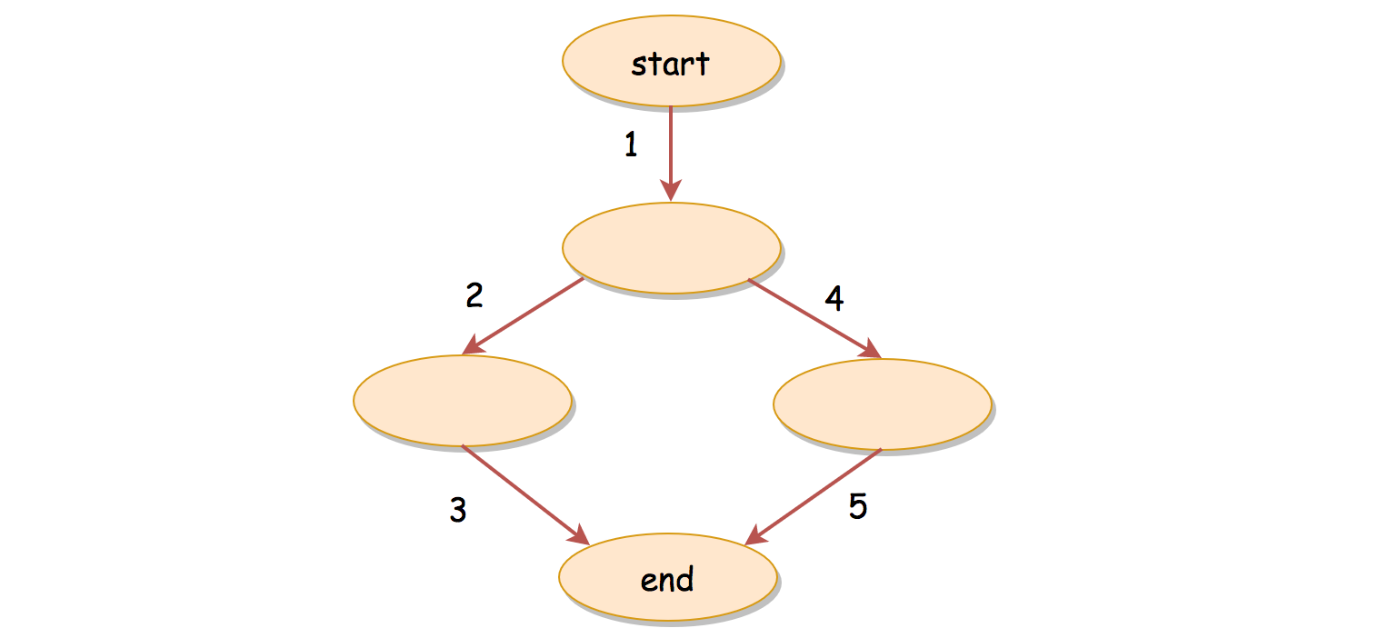
For more details about how to implement a backtracking algorithm, one can refer to our [Explore card](https://leetcode.com/explore/learn/card/recursion-ii/472/backtracking/).

**Intuition**

Specifically, for this problem, we could assume ourselves as an agent in a game, we can explore the graph one step at a time.

At any given node, we try out each neighbor node *recursively* until we reach the target or there is no more node to hop on. By trying out, we mark the choice before moving on, and later on we reverse the choice (*i.e.* backtrack) and start another exploration.

To better demonstrate the above idea, we illustrate how an agent would explore the graph with the *backtracking* strategy, in the following image where we mark the order that each edge is visited.



**Algorithm**

The above idea might remind one of the Depth-First Search (**DFS**) traversal algorithm. Indeed, often the backtracking algorithm assumes the form of DFS, but with the additional step of *backtracking*.

And for the DFS traversal, we often adopt the **recursion** as its main form of implementation. With recursion, we could implement a backtracking algorithm in a rather intuitive and concise way. We break it down into the following steps:

* Essentially, we want to implement a recursive function called backtrack(currNode, path) which continues the exploration, given the current node and the path traversed so far.
  + Within the recursive function, we first define its base case, *i.e.* the moment we should terminate the recursion. Obviously, we should stop the exploration when we encounter our target node. So the condition of the base case is currNode == target.
  + As the body of our recursive function, we should enumerate through all the neighbor nodes of the current node.
  + For each iteration, we first mark the choice by appending the neighbor node to the path. Then we *recursively* invoke our backtrack() function to explore *deeper*. At the end of the iteration, we should reverse the choice by popping out the neighbor node from the path, so that we could start all over for the next neighbor node.
* Once we define our backtrack() function, it suffices to add the initial node (*i.e.* node with index 0) to the path, to *kick off* our backtracking exploration.

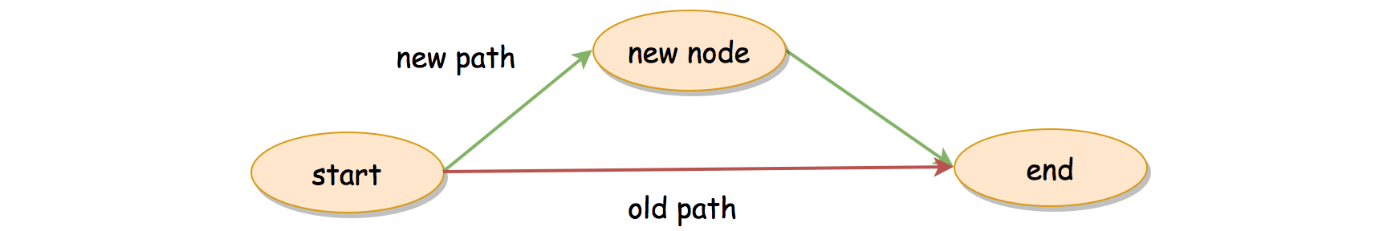
**Complexity Analysis**

Let N*N* be the number of nodes in the graph.

First of all, let us estimate how many paths there are at maximum to travel from the Node 0 to the Node N-1 for a graph with N*N* nodes.

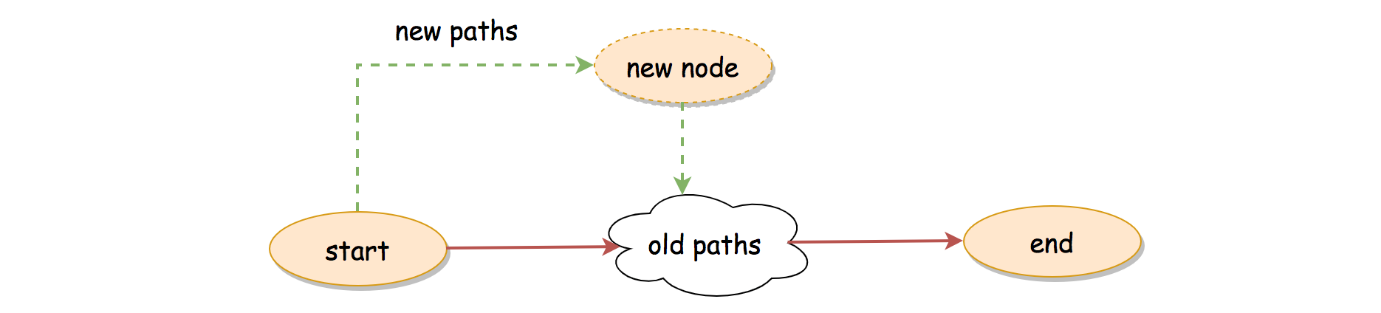
Let us start from a graph with only two nodes. As one can imagine, there is only one single path to connect the only two nodes in the graph.

Now, let us add a new node into the previous two-nodes graph, we now have two paths, one from the previous path, the other one is bridged by the newly-added node.



If we continue to add nodes to the graph, one insight is that every time we add a new node into the graph, the number of paths would **double**.

With the newly-added node, new paths could be created by preceding all previous paths with the newly-added node, as illustrated in the following graph.



As a result, for a graph with N*N* nodes, at maximum, there could be \sum\_{i=0}^{N-2}{2^i} = 2^{N-1} - 1∑*i*=0*N*−2​2*i*=2*N*−1−1 number of paths between the starting and the ending nodes.

* Time Complexity: \mathcal{O}(2^N \cdot N)O(2*N*⋅*N*)
  + As we calculate shortly before, there could be at most 2^{N-1} - 12*N*−1−1 possible paths in the graph.
  + For each path, there could be at most N-2*N*−2 intermediate nodes, *i.e.* it takes \mathcal{O}(N)O(*N*) time to build a path.
  + To sum up, a **loose** upper-bound on the time complexity of the algorithm would be (2^{N-1} - 1) \cdot \mathcal{O}(N) = \mathcal{O}(2^N \cdot N)(2*N*−1−1)⋅O(*N*)=O(2*N*⋅*N*), where we consider it takes \mathcal{O}(N)O(*N*) efforts to build each path.
  + It is a loose uppper bound, since we could have overlapping among the paths, therefore the efforts to build certain paths could benefit others.
* Space Complexity: \mathcal{O}(2^N \cdot N)O(2*N*⋅*N*)
  + Similarly, since at most we could have 2^{N-1}-12*N*−1−1 paths as the results and each path can contain up to N*N* nodes, the space we need to store the results would be \mathcal{O}(2^N \cdot N)O(2*N*⋅*N*).
  + Since we also applied *recursion* in the algorithm, the recursion could incur additional memory consumption in the function call stack. The stack can grow up to N*N* consecutive calls. Meanwhile, along with the recursive call, we also keep the state of the current path, which could take another \mathcal{O}(N)O(*N*) space. Therefore, in total, the recursion would require additional \mathcal{O}(N)O(*N*) space.
  + To sum up, the space complexity of the algorithm is \mathcal{O}(2^N \cdot N) + \mathcal{O}(N) = \mathcal{O}(2^N \cdot N)O(2*N*⋅*N*)+O(*N*)=O(2*N*⋅*N*).

Approach 2: Top-Down Dynamic Programming

**Intuition**

The backtracking approach applies the paradigm of divide-and-conquer, which breaks the problem down to smaller steps. As one knows, there is another algorithm called ***Dynamic Programming*** (DP), which also embodies the idea of divide-and-conquer.

As it turns out, we could also apply the DP algorithm to this problem, although it is less optimal than the backtracking approach as one will see later.

More specifically, we adopt the **Top-Down** DP approach, where we take a *laissez-faire* strategy assuming that the target function would work out on its own.

Given a node currNode, our target function is allPathsToTarget(currNode), which returns all the paths from the current node to the target node.

The target function could be calculated by iterating through the neighbor nodes of the current node, which we summarize with the following *recursive* formula:

\forall \text{nextNode} \in \text{neighbors}(\text{currNode}), \\ \\ \text{allPathsToTarget}(\text{currNode}) = \{ \text{currNode} + \text{allPathsToTarget}(\text{nextNode}) \}∀nextNode∈neighbors(currNode),allPathsToTarget(currNode)={currNode+allPathsToTarget(nextNode)}

The above formula can be read intuitively as: *"the paths from the current node to the target node consist of all the paths starting from each neighbor of the current node."*

**Algorithm**

Based on the above formula, we could implement a DP algorithm.

* First of all, we define our target function allPathsToTarget(node).
  + Naturally our target function is a recursive function, whose base case is when the given node is the target node.
  + Otherwise, we iterate through its neighbor nodes, and we invoke our target function with each neighbor node, *i.e.* allPathsToTarget(neighbor)
  + With the returned results from the target function, we then prepend the current node to the downstream paths, in order to build the final paths.
* With the above defined target function, we simply invoke it with the desired starting node, *i.e.* node 0.

Note that, there is an important detail that we left out in the above step. In order for the algorithm to be fully-qualified as a DP algorithm, we should **reuse** the intermediate results, rather than re-calculating them at each occasion.

Specially, we should **cache** the results returned from the target function allPathsToTarget(node), since we would encounter a node multiple times if there is an overlapping between paths. Therefore, once we know the paths from a given node to the target node, we should keep it in the cache for reuse. This technique is also known as ***memoization***.

**Complexity Analysis**

Let N*N* be the number of nodes in the graph. As we estimated before, there could be at most 2^{N-1}-12*N*−1−1 number of paths.

* Time Complexity: \mathcal{O}(2^N \cdot N)O(2*N*⋅*N*).
  + To estimate the overall time complexity, let us start from the last step when we prepend the starting node to each of the paths returned from the target function. Since we have to copy each path in order to create new paths, it would take up to N*N* steps for each final path. Therefore, for this last step, it could take us \mathcal{O}(2^{N-1} \cdot N)O(2*N*−1⋅*N*) time.
  + Right before the last step, when the maximal length of the path is N-1*N*−1, we should have 2^{N-2}2*N*−2 number of paths at this moment.
  + Deducing from the above two steps, again a **loose** upper-bound of the time complexity would be \mathcal{O}(\sum\_{i=1}^{N}{2^{i-1}\cdot i}) = \mathcal{O}(2^N \cdot N)O(∑*i*=1*N*​2*i*−1⋅*i*)=O(2*N*⋅*N*)
  + The two approach might have the same asymptotic time complexity. However, in practice the DP approach is slower than the backtracking approach, since we copy the intermediate paths over and over.
  + Note that, the performance would be degraded further, if we did not adopt the memoization technique here.
* Space Complexity: \mathcal{O}(2^N \cdot N)O(2*N*⋅*N*)
  + Similarly, since at most we could have 2^{N-1}-12*N*−1−1 paths as the results and each path can contain up to N*N* nodes, the space we need to store the results would be \mathcal{O}(2^N \cdot N)O(2*N*⋅*N*).
  + Since we also applied *recursion* in the algorithm, it could incur additional memory consumption in the function call stack. The stack can grow up to N*N* consecutive calls. Therefore, the recursion would require additional \mathcal{O}(N)O(*N*) space.
  + To sum up, the space complexity of the algorithm is \mathcal{O}(2^N \cdot N) + \mathcal{O}(N) = \mathcal{O}(2^N \cdot N)O(2*N*⋅*N*)+O(*N*)=O(2*N*⋅*N*).