Graph Searches - Unweighted Graphs

CMPUT 275 - Winter 2018

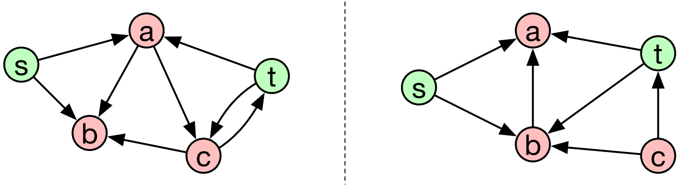
University of Alberta

Notation

See the file **Some Terms About Sets and Graphs** on eClass for notation, like $a \in X$.

Basic Problem

Consider two distinct vertices[1] s, t in a graph $G = (V; E)$.



Is there an $s - t$ path in G?

Yes in the first picture: $[s, a, c, t]$. **No** in the second.

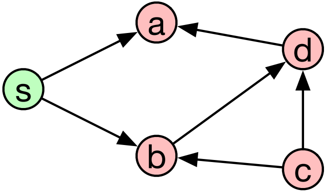
[1] Tip: **vertex** - singular, **vertices** - plural

We will actually solve a more general problem.

Graph Reachability Problem

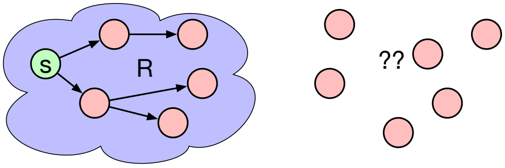
Input: A graph $G = (V; E)$ (can be directed or undirected). A particular vertex $s \in V$.

Output: The subset of **all** vertices $R \subseteq V$ reachable from s.



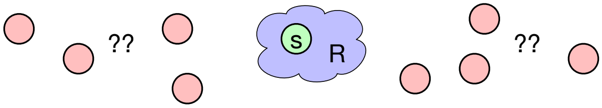
$R = \{s, a, b, d\}$.

The algorithm will maintain a subset R with the property that **every vertex in R can be reached by a path of vertices in R**.



Eventually R will grow to be all vertices reachable from s.

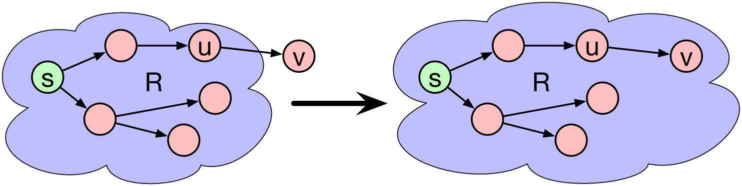
Initially: $R = \{s\}$
Because we know we can reach s from s with a trivial path: $[s]$.



How To Grow R?

Observation Suppose there is some $uv \in E$ with $u \in R$ and $v \notin R$.

We can reach v from s by following the $s - u$ path (which exists by the invariant for R) and then the edge uv.



Adding v to R maintains the invariant: v is reachable from s using vertices only in the new R.

Repeat until no edge exits R.

Summary/Pseudocode

Input: A graph $G = (V; E)$. A vertex $s \in V$.
Output: The set of vertices reachable from s.

Notation: $x \leftarrow y$ means x gets assigned y. Most pseudocode avoids using $=$, we will too.

Algorithm 1 Basic Reachability Algorithm

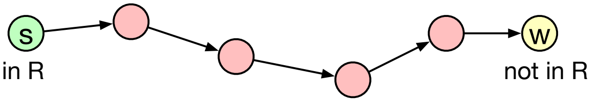
$R \leftarrow \{s\}$
while some edge $uv \in E$ has $u \in R, v \notin R$ **do**
    add v to R (i.e. $R \leftarrow R \cup \{v\}$)
return R

Does This Work?

We already discussed why the invariant holds: if v is added to R then it can be reached from s using only vertices in R.

But, when the algorithm finishes is R truly all reachable vertices?

Proof by Contradiction
Suppose not: suppose there is some $w \notin R$ reachable from s. Then there is a path $[s, v_1, v_2, \ldots, v_{k-1}, w]$ in G.



Note $s \in R, w \notin R$. So some **edge** (u, v) on the path has $u \in R, v \notin R$.
But then the algorithm would not have terminated!

Running Time

$R \leftarrow \{s\}$
while some edge $uv \in E$ has $u \in R, v \notin R$ **do**
    Add v to R
return R

Number of iterations: $O(|V|)$
Because each vertex can only be added to R once.

Running time per iteration: $O(|E|)$

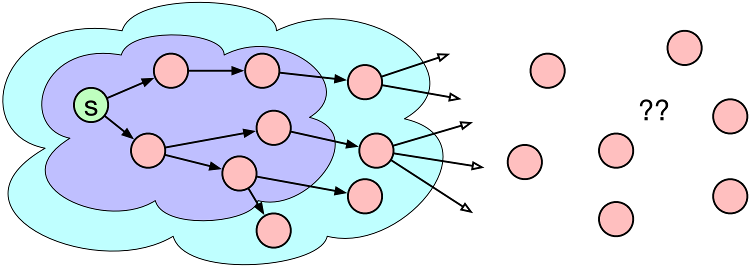
Running time bound: $O(|V| \cdot |E|)$.

The graph of Edmonton has hundreds of thousands of edges and vertices. **This is too slow!**

Improvements

For each v added to R, we only need to examine the edges exiting v once.

So keep track of an **open/unexplored set**: the vertices that have been reached but have not had their neighbours considered yet.



$R \leftarrow \{s\}$
$U \leftarrow \{s\}$
while U is not empty **do**
    pick some $u \in U$, remove it from U
    for each neighbour v of u **do**
        if $v \notin R$ **then** add v to U and R
return R

Here U is the vertices of R whose neighbours have not yet been examined.

Running time: Consider an iteration with, say, vertex u. The running time is $O(\#$ neighbours of $u)$. Thus, the total time is $O(|E|)$.

Now implement this!

Running Time Problem

Wait!
Our current implementation of the graph class has the `neighbours` method running in $O(|E|)$ time.

So each of the $O(|V|)$ iterations of the improved algorithm still takes $O(|E|)$ time. We are back to $O(|V| \cdot |E|)$ time. ☹

Idea: Change the internal implementation of the graph class to support this function faster!

Rather than storing a set of vertices and list of edges, just store a single dictionary `adj` so `adj[v]` is the list of all neighbours of `v`. This way, we can just return this list in $O(1)$ time!

This is called the **adjacency list** representation of the graph.

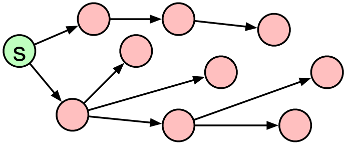
Coding Break: Do it!

Now we are back to $O(|E|)$ running time. Phew!

Recovering a Path

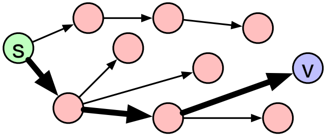
What if we want an actual path (list of vertices) from s to some reachable vertex?

The search builds a "search tree". Consider R plus every edge uv used in the search to include a vertex v in R.



We can store the search tree in a dictionary that maps each $v \in R$ to its *predecessor* u on the search (storing s at key s).

Do It: Replace R with a dictionary storing the predecessor in this way.



To recover an actual path to some $v \in R$, we then just crawl back through the tree from v until we reach s.

Algorithm 2 Recovering an $s - v$ path from a search tree *reached*

$path \leftarrow [v]$ # a list
while $v \neq s$ **do**
  $v \leftarrow reached[v]$
  append v to *path*
reverse *path*
return *path*

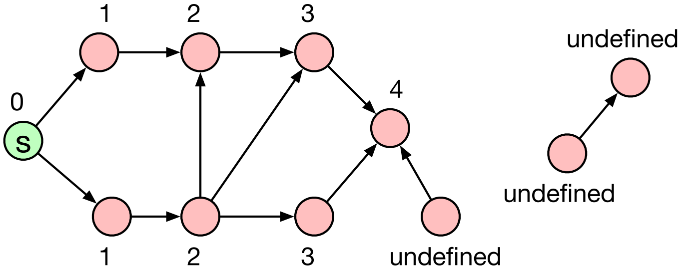
Running Time: $O(len(path))$

Just $s - t$ **Paths**? If you only care about a path to a specific vertex, you can stop the search as soon as it is reached.

Undirected graphs? Build the directed graph with both uv and vu for each undirected edge uv. Then run the search.

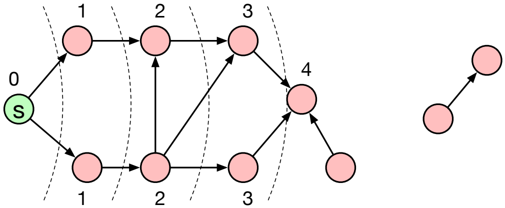
What's Next?

Shortest Paths. Say that an $s - t$ path is a *shortest* path if it has the minimum number of edges of an $s - t$ path.



Bright idea! Process vertices in the same order they were added. They will come out in nondecreasing order of distance!

Queues: Instead of a set U of unvisited vertices, use a queue and always remove the front of the queue. In Python, we can use a `deque`.



- After processing the start (the only distance-0 vertex) all distance-1 vertices are in the queue.
- After processing all distance-1 vertices all distance-2 vertices are in the queue.
- and so on ...

Breadth-First Search

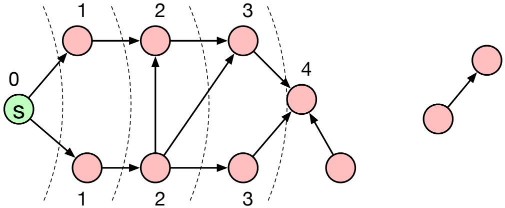
Algorithm 3 Breadth-First Search With Start Vertex s

$R[s] \leftarrow s$
$Q \leftarrow$ a new queue/deque containing only s
while Q is not empty **do**
    $u \leftarrow \texttt{pop}(Q)$
    for each neighbour v of u **do**
        if v is not in R **then**
            $R[v] \leftarrow u$
            $\texttt{push}(Q, v)$
return R

Running Time: $O(|E|)$. Each edge (u, v) is considered at most one among all executions of the inner for loop.

Depth-First Search

While **breadth-first search** is great for finding shortest paths, it's not really how you would navigate a graph if you were traversing it.



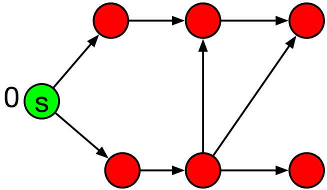
A more *natural* search for a real agent that has to walk around a graph is a **depth-first search**.

A depth-first search is a recursive search. When it reaches a vertex u it recursively explores each neighbour in turn.

Mark off each vertex that is visited so you don't recursively explore it twice.

Example - Vertices Numbered in Order of First Visit

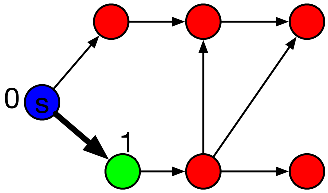
- green - current vertex
- blue - visited and currently recursively searching some neighbour
- black - visited and all neighbours have been recursively searched
- red - not yet visited



Start the search from s.

Example - Vertices Numbered in Order of First Visit

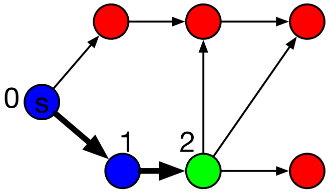
- green - current vertex
- blue - visited and currently recursively searching some neighbour
- black - visited and all neighbours have been recursively searched
- red - not yet visited



Follow the bottom arrow.

Example - Vertices Numbered in Order of First Visit

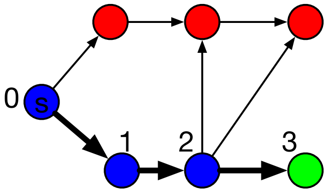
- green - current vertex
- blue - visited and currently recursively searching some neighbour
- black - visited and all neighbours have been recursively searched
- red - not yet visited



Follow the only edge.

Example - Vertices Numbered in Order of First Visit

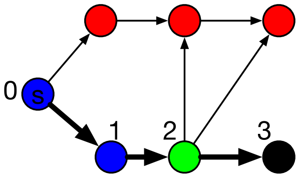
- green - current vertex
- blue - visited and currently recursively searching some neighbour
- black - visited and all neighbours have been recursively searched
- red - not yet visited



Suppose we followed this edge from vertex #2.

Example - Vertices Numbered in Order of First Visit

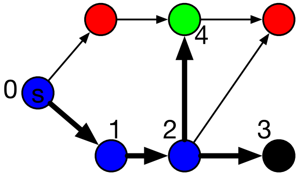
- green - current vertex
- blue - visited and currently recursively searching some neighbour
- black - visited and all neighbours have been recursively searched
- red - not yet visited



No edges exiting #3, backtrack to #2 and try another edge.

Example - Vertices Numbered in Order of First Visit

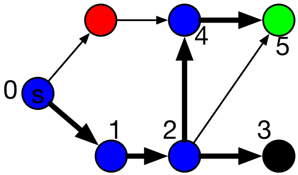
- green - current vertex
- blue - visited and currently recursively searching some neighbour
- black - visited and all neighbours have been recursively searched
- red - not yet visited



Suppose we picked this edge from #2.

Example - Vertices Numbered in Order of First Visit

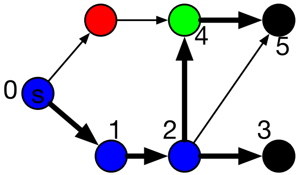
- green - current vertex
- blue - visited and currently recursively searching some neighbour
- black - visited and all neighbours have been recursively searched
- red - not yet visited



Follow the only edge from #4.

Example - Vertices Numbered in Order of First Visit

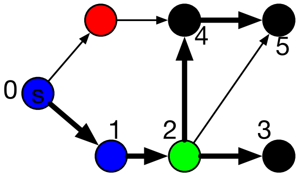
- green - current vertex
- blue - visited and currently recursively searching some neighbour
- black - visited and all neighbours have been recursively searched
- red - not yet visited



No edges from #5, backtrack to #4..

Example - Vertices Numbered in Order of First Visit

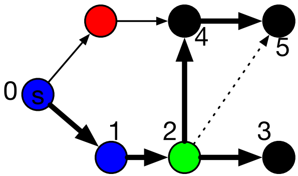
- green - current vertex
- blue - visited and currently recursively searching some neighbour
- black - visited and all neighbours have been recursively searched
- red - not yet visited



No unexplored edges from #4, backtrack to #2.

Example - Vertices Numbered in Order of First Visit

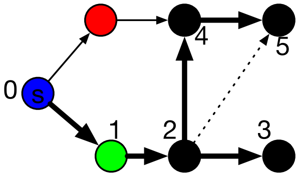
- green - current vertex
- blue - visited and currently recursively searching some neighbour
- black - visited and all neighbours have been recursively searched
- red - not yet visited



The edge $2 \rightarrow 5$ has not yet been explored from #2. But #5 is visited, so don't cross the edge.

Example - Vertices Numbered in Order of First Visit

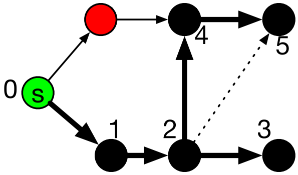
- green - current vertex
- blue - visited and currently recursively searching some neighbour
- black - visited and all neighbours have been recursively searched
- red - not yet visited



No unexplored edges from #2, backtrack to #1.

Example - Vertices Numbered in Order of First Visit

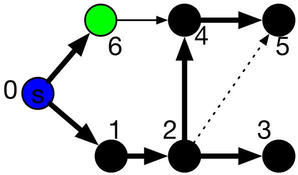
- green - current vertex
- blue - visited and currently recursively searching some neighbour
- black - visited and all neighbours have been recursively searched
- red - not yet visited



No unexplored edges from #1, backtrack to #0 (a.k.a. s).

Example - Vertices Numbered in Order of First Visit

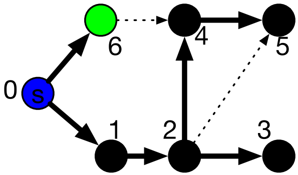
- green - current vertex
- blue - visited and currently recursively searching some neighbour
- black - visited and all neighbours have been recursively searched
- red - not yet visited



Explore the other edge out of #0.

Example - Vertices Numbered in Order of First Visit

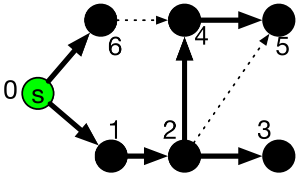
- green - current vertex
- blue - visited and currently recursively searching some neighbour
- black - visited and all neighbours have been recursively searched
- red - not yet visited



The only edge out of #6 reaches a visited vertex, so don't explore it.

Example - Vertices Numbered in Order of First Visit

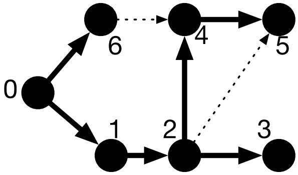
- green - current vertex
- blue - visited and currently recursively searching some neighbour
- black - visited and all neighbours have been recursively searched
- red - not yet visited



Backtrack to #0 as there are no more edges from #6.

Example - Vertices Numbered in Order of First Visit

- green - current vertex
- blue - visited and currently recursively searching some neighbour
- black - visited and all neighbours have been recursively searched
- red - not yet visited



No unexplored edges from #0 (the start), so we are done!

Comments About The Example

The green, blue, and black vertices are the ones that have been visited.

The green and blue vertices always form a path starting at s and ending at the green vertex. This path is also the current recursion stack with the green vertex being at the top of the stack.

Python Note: There is a maximum recursion depth limit of 1000 calls that is imposed by the Python interpreter. But this can be changed, example:

```
sys.setrecursionlimit(25000)
```

Depth-First Search

Let R be a *global* dictionary, initially empty.

Algorithm 4 Depth-First Search(u, *prev*)

```
if u is in R then
    return
R[u] ← prev
for each neighbour v of u do
    Depth-First Search(v, u)
```

The initial recursive call should be with the start vertex for both arguments.

Running Time: $O(|E|)$. Each edge (u, v) is considered at most one among all recursive calls.

Summary

Breadth-First Search

- A search using a queue to process the vertices.
- Finds shortest paths (min # of edges) to all reachable vertices.

Depth-First Search

- Uses recursion to search the graph.
- Usually does not find shortest paths.
- Can have a real "agent" traverse an undirected graph using a depth-first search.

Other Applications of Depth-First Search

In Linear Time, i.e. $O(|V| + |E|)$

- **Topologically sort** a directed, acyclic graph (worksheet).
- Find all **bridges** of an undirected graph (an edge whose removal disconnects the graph).
- Find all **cut vertices** of an undirected graph (a vertex whose deletion disconnects the graph)
- Find **strongly connected components** of a directed graph in linear time.

Look these terms up if you are interested!