CSI 310: Lecture 26 (finishing graph traversal)
Read it to learn the algorithm and code it yourself.

to a class maze object.

DSO's code applies to a class Graph object, your code applies to (or resides in)

unless you really know what you are doing.

DO NOT COPY AND MODIFY THIS CODE FOR PROJECT 5 parts 2 and 3!

Caution: (in response to questions after last lecture)
template <class Process, class Item, class SizeType>

void depth-first(Process fi, Graph<Item> & G, SizeType start);

template <class Process, class Item, class SizeType>

void breadth-first(Process fi, Graph<Item> & G, SizeType start);

helper function which expresses most of the algorithm.

Depth first search is invoked by a non-recursive non-member function that calls a

non-member C++ function.

Breadth first search is invoked and expressed with one non-recursive
```cpp
for(int i=0; i<E.size(); i++) marked[i]=false;

std::fill_n(marked, E.size(), false);
std::queue<set<std::size_t>> vertex-queue;
std::set<set<std::size_t>> connections;
bool marked=E.MAX;

std::size_t start; // type used for vertex numbers

type Item; // Item class

process Item first

template <class process, class Item, class SizeType>
```
assert (start < E::size());
markE[markE[start] = true;
if (E[markE[start]]) {
vertex-queue.push(start);
}
```cpp
{ 
  { 
    "vertex-queue.push.\(\langle V \rangle\); 
    if \(E[V]\) \n      \(\text{is processed.}\) 
    else { 
      [\(\text{is marked.}\) \n        \(\text{if true:}\) 
        \(\text{[is not in vertex-queue.}\) 
        \(\text{INV: N is not in vertex-queue.}\) 
      } 
    } 
  } 
} 

(\text{if marked}) 
\(\text{if it is a neighbor of } V, \text{ call it N.}\) 
\(\text{INV: N is not in vertex-queue.}\) 
(\text{for } it = \text{connections.begin}(\); \text{it } = \text{connections.end}(\); \text{it }++) 
\text{std::set<std::size_t>::iterator it;} 
\text{vertex-queue.pop.}\) 
\(\text{vertex \& is popped.}\) 
\(\text{vertex-queue.front.} \text{neighbors(vertex-queue.front.)}\) 
} 
```
{ // INV: queue is empty, all reachable vertices are marked.
    while (vertex-queue.empty()) {
    }
template <class Process, class Item, class SizeType>
void depth-first-process<Item, Graph, Item, Graph> process
(size_type start)
that the search has just arrived at.  

The vertex \( v \) is an unmarked vertex

//

this search, otherwise marked[v] is false.

//

For each vertex \( x \), marked[x] is true if \( x \) has already been visited.

//

For a labeled graph that is being traversed by a depth-first search

//

Preconditions:

void rec-dfs(Processor<Process<T, Graph<Item, G, Sz extends P, x, P>, Siz> P, boolean marked[])
Library facilities used: stdlib, graph.h, set

// marked[x] to true.

the search and each such vertex x has also been marked by set

// The function f has been applied to the label of each vertex visited.

// from v via a path of unmarked vertices.

// that can be reached

// and beyond to all the vertices

// The depth-first search of g has been continued through vertex a

// Postconditions:
return; // DONE, return from the activation that marked vertex.

{ 
  rec-dfs(f, G, \*f, marked)
  
  if (marked[f])
  
  for (\[i\] = connections.begin(); \[i\] != connections.end(); ++\[i\])
  
  std::set\<std::size_t\>::iterator \[i\];

  std::set\<std::size_t\>::size_t neighbors = G.neighbors(\[i\]);

  std::set\<std::size_t\>::connections = G.neighbors(\[i\]);

  for (\[j\] = 0; \[j\] < neighbors; ++\[j\])

  \[j\](\[G\][\[i\]])

  \[f\] = \[i\]

  TWO: process the label of vertex \[i\] with the function \[f\]

  \[f\][\[i\]]

  \[G\][\[i\]]

  ONE: Mark vertex \[i\] = true;

  \[marked[\[i\]] = true; \]

  } // void rec-dfs(Process f, GraphItem G, \*G, size_t f, Item g, Type g, bool marked, \*marked)

} // void rec-dfs(Process f, GraphItem G, \*G, size_t f, Item g, Type g, bool marked, \*marked)
Fact: Depth-First Traversal of the entire network will always find a cycle this way if there is one. 

There is one cycle this way if there is one.

Found already, but not in the linked list of Part 1.

Parts of graph not traversed yet.

Start

The first time

Tree of edges used to find a vertex

Not traversed parts of graph

Path of vertices of current activations

The first time

Edge finding a vertex found already, but not in the linked list of Part 1.
Theorem (proved in the CS1403/503 books): Network $\mathcal{F}$ has one or more cycles if

\begin{verbatim}
{ 

dfs-rec ( \cdot, E, \Lambda ) 

initialize-list \Lambda to (\Lambda )

while ( \Lambda is some unmarked vertex)

How to test if a directed network $\mathcal{F}$ (i.e., graph) has one or more (directed) cycles:

execution stack:

Currently running activation to $T_0$ a vertex of an activation currently in the

and only if an edge was found during dfs that went FROM the vertex of the

\end{verbatim}
/* end of for */ {
/* end of else */ {

count > Hey. this network has a cycle

} if (*tt) is in L

else

{ remove (*tt) from L list L;
  rects[tt] = 0;  // mark marked
  append (*tt) to L list L;
  for (tt = connections.begin(); tt != connections.end(); ++tt) {
    for (it = neighbors.begin(); it != neighbors.end(); ++it) {
      if (marked[v] = true;
DISTANCE from the start vertex.

Breadth-first traversal does classify vertices according to their distance. Therefore might be a cycle among these vertices, but the traversal can't find it.

\[ \text{Dist} = 0, 1, 2 \]