Introduction to Computer Science and Programming for Astronomers
Lecture 7.

István Szapudi

Institute for Astronomy
University of Hawaii

February 26, 2014
Outline

1. Reminder

2. Graphs
   - Introduction–Representation
   - Connectivity
   - Spanning Trees
   - Brute Force and Backtracking
We have seen that using properties of the data one can do a faster sort: e.g. bucket sort.

We have defined order statistics, and concluded that selection of the $k$-th element is needed.

We have found that we can do that without sorting, with a linear, randomized algorithm, quickSelect.

Note that there are deterministic selection algorithms

Now we move to basic algorithms with graphs.
Packing Problems

- Packing problem: pack $N$ things into a box. However, there exist pairs which mutually exclude each others.
- Choose the most people among $N$ persons that none of them knows the other.
- A truck goes from $A$ to $B$. Along its drive, there are places $A_1, \ldots, A_N$, and $B_1 \ldots B_N$ (not necessarily in order). The truck needs to deliver goods from $A_i$ to $B_i$. Which orders should it accept to deliver the most goods?
A graph consists of a collection of vertices and edges. The vertices typically represent a collection of objects, the edges some sort of connection between the objects. Directed graph: edges have a direction, otherwise undirected. Edges and vertices can have weights (numbers) associated with them. Notation $n = V(G)$ for the number of vertices, and $m = E(G)$ for edges.
Graphs represent extremely abstract and at the same time common ideas:

- Vertices: the astronomers of the world. Edges: each people who have written papers together are connected with an edge.
- Directed graph: still astronomers: for edges use letters of recommendations from the person who has written the letter to the other.
- A galaxy catalog, the vertices are galaxies, the edges connect galaxies closer then $5h^{-1}\text{Mpc}$. (Remember our very first problem!)
- A galaxy catalog, every single galaxy is connected with every other one, and the edge has a weight proportional to the distance of the pairs (A full, weighted graph).
Object Oriented representation: we store the list of vertices, say \( A, B, C, D \), and the edges as pairs of vertices \((A, B), (A, C), \ldots \), etc.

The total space is \( \mathcal{O}(n + m) \).

In `python` this can be done e.g., with two lists:

- \( \text{vertices} = [\text{'A'}, \text{'B'}, \text{'C'}, \text{'D'}] \)
- \( \text{edges} = [[\text{'A'}, \text{'B'}], [\text{'A'}, \text{'C'}], \ldots] \)

Many operations in this representation involve scanning the whole list of edges, i.e. take \( \mathcal{O}(m) \) time.
Adjacency list: for each edge we store which edges it is connected to. \( A: B, C, D, \) etc.

This is still \( O(m + n) \), each edge is listed twice.

In Python you can use dictionaries:

\[
g = \{ 'A': [ 'A', 'B', 'C' ], \ldots \}
\]

The slowest operation is testing whether a pair of vertices is connected by an edge.

In principle you need to scan the whole list, however, in the above Python representation the dictionary is automatically hashed, so it is quite fast. In other languages you can use a hashed representation, or presorting the adjacency list, and use binary search.
A combination of the previous two representations.

We start with object oriented representation, and add a pointer for each vertex to all the edges incident to it.

This takes more space, but still $O(m + n)$.

In python

```python
ev = {
    'A': [0, 1, ...],
    'B': [0, 2, ...], ...
}
```

where

```python
e = [
    ['A', 'B'], ['A', 'C'], ['B', 'C'], ...
]```
The Adjacency Matrix has $M_{ij} = 1$ if vertices $i, j$ are connected, 0 otherwise.

For undirected graphs it is a symmetric matrix.

Storage is $O(n^2)$.

Finding neighbors takes $O(n)$ time, but testing for the existence for an edge is $O(1)$.

In $M^k$ there is a 1 iff the two vertices are connected with $k$ edges.
It is an $n \times m$ matrix, indexed by the vertices and edges.

It has a $M_{ij} = 1$ where a particular vertex $i$ and edge $j$ are incident, otherwise 0.

For a directed graph the Incidency Matrix will contain 1, −1’s.

This is not symmetric matrix (not even square).
Try to build a list `reachable`, the list of nodes reachable from an arbitrary node.

We create a list `exploreList`. We will explore all items in that list. Initially both lists contain an arbitrary node.

We take an element from the explore list, explore its neighbours. If they are not included in the reachable list, we store them as reachable, but also add them to the items to be explored.

If in the end the reachable list contains all the vertices of the original graph, it is connected, otherwise not.
def testConnected(G):
    """
    test the connectedness of a graph G.
    G is represented by adjacency list (dictionary)
    {'A': ['B', 'C', ...], ...}
    """
    reachableList = [G.keys()[0]]
    exploreList = [G.keys()[0]]
while exploreList != []:
    v = exploreList.pop()
    for v1 in G[v]:
        if reachableList.count(v1) == 0:
            reachableList.append(v1)
            exploreList.append(v1)

if len(reachableList) < len(G):
    return 'disconnected'
else:
    return 'connected'
It is a waste searching `reachableList` if the vertex `v1` is in there already. Solution: we can use a lookup table.

```python
def testConnectedL(G):
    """
    uses an extra lookup to speed up searching for v1
    """

    # initially none of the vertices are explored
    explored = {}
    for v in G:
        explored[v] = 0
```
testConnected: Cont’d

while exploreList != []:
    v = exploreList.pop()
    for v1 in G[v]:
        if explored[v1] == 0:
            reachableList.append(v1)
            exploreList.append(v1)
            explored[v1] = 1

if len(reachableList) < len(G):
    return ‘disconnected’
else:
    return ‘connected’
Connectivity: Analysis

- The outer loop is executed for each vertex, i.e. for \( n \) times for a connected graph.
- The inner loop is executed \( 2m \) times, as much as the total length of the adjacency lists.
- Just like with the bucket sort, we don’t need to multiply with the outer loop, just add up the total times each statement is executed.
- All in all, this gives \( \mathcal{O}(n + m) \).
- For object oriented representation, the inner loop with scan through each edge, giving \( \mathcal{O}(nm) \).
- In the adjacency matrix representation, the inner loop would involve looking at a single row of the matrix: \( \mathcal{O}(n^2) \).
Spanning Trees

- Remember that a tree is a graph without loops.
- In a tree there is only path from any two point to another.
- Spanning tree of a connected graph is a subgraph of the original graph, which is also a (connected) tree.
- Spanning tree is not unique (sometimes we want the minimal spanning tree in a weighted graph).
- The previous algorithm can be modified to provide a spanning tree.
- There are at least three (Kruskal, Prim, Boruvka) famous algorithms for minimal spanning trees.
def spanningTreeDFS(G):
    reachableList = [G.keys()[0]]
    exploreList = [G.keys()[0]]
    spanningTree = {G.keys()[0]:[]}
    explored = {}
    for v in G:
        explored[v] = 0
        explored[G.keys()[0]] = 1
spanningTreeDFS Cont’d

```python
while exploreList != []:
    v = exploreList.pop()
    for v1 in G[v]:
        if explored[v1] == 0:
            reachableList.append(v1)
            exploreList.append(v1)
            explored[v1] = 1
            spanningTree[v1] = []
            spanningTree[v].append(v1)

    if len(reachableList) < len(G):
        return 'Error: this graph is disconnected!'

else:
    return spanningTree
```
We have not payed attention to how in what order we put things on the `exploreList`.

In the above example, it functioned as a stack: we pushed and popped things from the same end.

This results in a “depth first” traversal of the graph.

The opposite choice, where `exploreList` functions as a queue, we get a “breadth first” version.

The only change is `v = exploreList.pop(0)`.
DFS vs BFS Example:

Initial graph:

\[ g = \{1: [2, 3, 4], 2: [5, 6, 7], 3: [8, 9], 4: [1, 9], 5: [2], 6: [2], 7: [2, 8], 8: [7, 3], 9: [3, 4]\} \]

Result of DFS spanning tree:

\[ \{1: [2, 3, 4], 2: [5, 6], 3: [8], 4: [9], 5: [], 6: [], 7: [], 8: [7], 9: []\} \]

Result of BFS spanning tree:

\[ \{1: [2, 3, 4], 2: [5, 6, 7], 3: [8, 9], 4: [], 5: [], 6: [], 7: [], 8: [], 9: []\} \]
The algorithm of algorithms
Try all possibilities

- The general packing problem has no good solution in general.
- The most general algorithm is to try all possibilities systematically.
- This can be very slow, but sometimes we have no other solution.
- Note: many such problems can be solved at least approximately with stochastic algorithms, which sample a random subset of possibilities.
- A slight variation on brute force is backtracking. It is the foundational algorithm of AI.
Finding a Path

- Given two vertices in a graph, find a path between the two.
- We start the path from first node, and find all nodes to which it is possible to get from that node.
- Then one by one we call the same algorithm for all of these nodes.
- This implementation is actually from Guido van Rossum, of python hall of fame...
def find_path(graph, start, end, path=[]):
    """
    find path from start to end, using backtracking
    """
    path = path + [start]
    if start == end:
        return path
    if not graph.has_key(start):
        return None
    return None
```python
for node in graph[start]:
    if node not in path:
        newpath = find_path(graph, node, end, path)
        if newpath: return newpath

return None
```
def find_all_paths(graph, start, end, path=[]):
    """
    don’t stop at the first path
    """
    path = path + [start]
    if start == end:
        return [path]
    if not graph.has_key(start):
        return []
    paths = []
    paths = [path]
find_all_path Cont’d

for node in graph[start]:
    if node not in path:
        newpaths =
            find_all_paths(graph, node, end, path)
        for newpath in newpaths:
            paths.append(newpath)

return paths
```python
find_shortest_path

Yet another modification

shortest = None
for node in graph[start]:
    if node not in path:
        newpath =
            find_shortest_path(graph, node, end, path)
        if newpath:
            if not shortest or
                len(newpath) < len(shortest):
                shortest = newpath

return shortest
```
There is no brilliant solution known for the original packing problem (finding the maximal set of disconnected vertices).

We can use the following observation. Let’s pick an arbitrary element $x \in G$. If $x$ is part of the maximal set $S$, then $S - x$ will be a maximal set for $G - A$, where $A$ denotes all the neighbours of $x$.

Conversely, if $x \not\in S$, $S$ is a maximal set for $S - x$ as well.

This means that we could check $S - x$ and $S - A - x$, and pick the better solution recursively.
def packWell(G):
    """
    backtracking solution to the packing problem: we want to choose the maximal set of vertices which are mutually disconnected.
    
simple branch and bound cut is demonstrated """

    if len(G) == 1:
        return [G.keys()[0]]
    if len(G) == 0:
        return []
packWell
\( G - x \)

# first look at G-x
gLoc = {}
gLoc.update(G)  #dictionary deep copy
x = gLoc.keys()[0]
A = gLoc.pop(x)

s1 = packWell(gLoc)
`packWell` Then $G - A - x$ with Branch and Bound

```python
gLoc.update(G)
x = gLoc.keys()[0]
A = gLoc.pop(x)
for a in A:
    if gLoc.has_key(a):
        gLoc.pop(a)

if len(gLoc) > len(s1)-1:
    s2 = packWell(gLoc)
else:
    return s1
```
if len(s1) >= len(s2)+1:
    return s1
else:
    return s2+[x]
We have introduced the notion of graphs.
We looked at how graphs are represented in the computer.
We looked at some of the most basic algorithms: connectivity, spanning trees, and finding a path.
We introduced yet another algorithmic paradigm, brute force or backtracking.
A very important speed-up for backtracking is the “branch and bound cut”.
Homework # 7

(E1) Consider an edge \((A, B)\) of a graph. Let’s ”pull vertex \(B\) onto \(A\)” to create a new graph with \(n - 1\) vertices. All edges which were connected to \(B\) are now connected to \(A\). If the original graph was connected, this new, smaller graph is also connected. Using this argument, a connected graph can be shrunked into 1 point, while a disconnected one cannot.

Use this observation and the Adjacency Matrix representation to write a new connectivity algorithm (Hint: if there is a 1 in the \(i\)-th position of the matrix, think through what would be the adjacency matrix of the graph, where the first vertex is pulled onto the \(i\)-th). What is the scaling of the resulting “topological” algorithm?
Homework # 7
continued

(E2) Measure and analyse the speed of the backtracking algorithm for the packing problem.

(E3) Store a weighted graph as a dictionary of dictionaries, e.g. \(g = \{ 'A': \{ 'B': 1, 'C': 2, \}, \ldots \}\), i.e. \(g[ 'A' ]\) will give a dictionary of edges connected, and \(g[ 'A' ][ 'B' ] = 1\) will be the weight of edge \((A, B)\). Modify the `find_shortest_path` algorithm to find the “cheapest” path from start to end, where the cost of a path is the sum of the length.