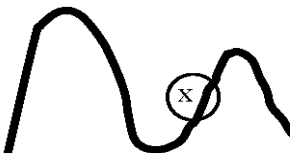
Greedy algorithms

A *greedy* algorithm follows the heuristic of making a locally optimum choice at each stage with the hope of reaching a global optimum.

You can understand this idea of greediness most easily with an *unsuccessful* greedy algorithm. Suppose you are standing at point `x` on the smaller of the two hills to the right, and you want to climb up to the highest point. The only information you have is the slope of the hill where you are standing. So, you go uphill at point `x`. You are using local information, hoping it will get you to the highest point. But it doesn't! Continue do that and it gets you only to the top of the smaller hill, not the larger one.



That's the *greediness* paradigm. Climb a hill using the local slope, hoping the highest point will be reached. In some cases it will, it others it will not.

A coin problem where a greedy algorithm works

The U.S. has these coins: half dollar (50 cents), quarter (25), dime (10), nickel (5), and penny (1). Here is an algorithm to calculate change for `n` cents, where `1 ≤ n`, using as few coins as possible. Since as few coins as possible should be used, the greediness comes from choosing at each stage the most expensive coin possible.

> As long as `n ≥ 50`, choose one half dollar and subtract `50` from `n`;
> If `n ≥ 25`, choose one quarter and subtract `25` from `n`;
> As long as `n ≥ 10`, choose a dime and subtract `10` from `n`;
> If `n ≥ 5`, choose a nickel and subtract `5` from `n`.
> Choose `n` pennies.

It can be proven that the chosen coins are always the fewest number possible. This greedy algorithm works. Millions of people use this algorithm every day in making change.

A coin problem where a greedy algorithm doesn't work

Suppose we have U.S. coins but we are out of nickels; the coins to choose from are the half dollar, quarter, dime, and penny. We still want to make change using the minimum number of coins possible. We use the same greedy algorithm as above but with the instruction about nickels deleted.

Suppose we ask for change of 30 cents. The greedy algorithm produces a quarter and 5 pennies. That's 6 coins. But instead one can use 3 dimes. The greedy algorithm doesn't work. There are two possible hills to climb; we start off on the wrong hill.

Dijkstra's shortest path algorithm is greedy —and it works

Dijkstra's shortest path problem is greedy. At each iteration, it chooses a node in the frontier set with minimum priority and moves it into the settled set. We *proved* that this greedy choice works! So this shortest path algorithm is an example of a successful greedy algorithm.

Adjacency matrix representation

One has to be careful because Java arrays start with 0. Here are two possibilities to deal with the difference between a matrix starting with row 1 and a Java array starting with index 0: (1) Declare the array of size [0..n][0..n] and don't use row 0 and column 0. (2) Give the nodes numbers in 0..n-1 instead of in 1..n.

Adjacency list representation

The adjacency list representation of a graph maintains a list or set of nodes. The list element for a node `u` contains a list of nodes that are adjacent to it, i.e. a list of nodes `v` such that (`u`, `v`) is an edge.

To the right is the adjacency list representation of the 4-node graph given above. Three directed edges leave node 1; therefore, the list for node 1 contains the sinks of these three edges: 4, 2, and 3, in no particular order. One directed edge leaves node 3, so the list for node 3 contains its sink, 2. The lists for nodes 2 and 4 are empty because no edges leave these nodes.

```
1 (4, 2, 3)
2 ()
3 (2)
4 ()
```

What data structures might we use in Java for the adjacency list representation? First, the list of nodes could be given in an array, an `ArrayList`, a `LinkedList`, a `Set` —in many different ways. Second, the element for each node can be a `List` of nodes or node numbers —e.g. an `ArrayList` or a `LinkedList`. The important point is that processing the edges leaving a node —or the sinks of those edges— should take time proportional to the length of the list. Later, will use this property in discussing the time to perform various operations on a graph.

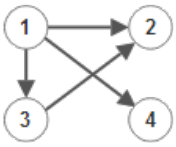
In an OO implementation of a graph. There would typically be classes `Graph`, `Node`, and `Edge`. Class `Graph` would have methods to retrieve (1) the `Nodes` as a `Set` or `List` and (2) the `Edges` as a `Set` or `List`. Class `Node` would have methods to retrieve (1) the `Edges` leaving the `Node` and (2) the `Nodes` at the other end of `Edges` leaving the `Node`. You can figure out what methods class `Edge` could have.

When deciding on an OO implementation of a graph, one must take into consideration the space requirements as well as the time requirements.

Space and time considerations

We are about to discuss the space and time requirements for the two graph representations. Some students are inclined to try to memorize the space and time requirements, and they will ask on the Piazza for the course (or whatever the discussion board is) if they forget. But this is plain rote memorization and not at all useful. Don't do it! Instead, understand the two representations of graphs and figure out the space and time requirements whenever required. It's not that difficult.

Consider a directed graph with n nodes and e edges. To the right, we again give the 4-node graph from above and its two representations, to make it easier for you to understand what is said below.



	1	2	3	4
1	0	1	1	1
2	0	0	0	0
3	0	1	0	0
4	0	0	0	0

1 (4, 2, 3)
2 ()
3 (2)
4 ()

Just which representation one uses depends on what kind of graph it is and how it will one used.

The table below gives the space requirements of each graph representation and gives the time complexities for two operations on the graph. Analysis to help you understand the entries in the table are given below he table.

	Adjacency list	Adjacency list for sparse graphs	Adjacency list for dense graphs	Adjacency matrix
Space	1: O(n+e)	2: O(n)	2: $O(n^2)$	3: $O(n^2)$
Enumerate all edges	4: O(n+e)	4: O(n)	4: $O(n^2)$	5: $O(n^2)$
Is there an edge (u, v)?	6: O(outdegree of node u)	6: O(outdegree of node u)	6: O(outdegree of node u)	7: O(1)

1: The adjacency list representation requires space O(n) for the list of nodes and, for each node, space proportional to the outdegree of the node for its adjacency list. Since there are *e edges*, the adjacency list representation requires O(n+e) space.

The n in O(n+e) is important. Suppose e is 0 —there are no edges. Then there are n empty lists of edges, each of which takes O(1) space, so the total space is O(n).

2: A sparse graph has O(n) edges, so the space requirement O(n+e) reduces to O(n). A dense graph has $O(n^2)$ edges, so the space requirement is $O(n^2)$.

3: The adjacency matrix has n^2 entries, so it takes space $O(n^2)$. But it requires only n^2 bits, while an adjacency list for a dense graph requires many more because each entry in a list is an integer and the list could be implemented as a linked list.

4: To enumerate all edges when an adjacency list is used requires looking at each of the n nodes and for each looking through the list of its edges. Since there are a total of e edges, this takes time O(n+e). For a sparse graph, with O(n) edges, this can be simplified to O(n). For a dense graph, with $O(n^2)$ edges, this can be simplified to $O(n^2)$.

5: Enumerating the edges when using an adjacency matrix requires looking at each entry of the n x n matrix.

6: When using an adjacency list, determining whether (u, v) is an edge may require looking at all entries in u's adjacency list.

7: When using an adjacency matrix, determining whether (u, v) is an edge requires looking only at entry m[u, v].