Algorithms/Greedy Algorithms

Top, Chapters: 1, 2, 3, 4, 5, 6, 7, 8, 9, A

In the backtracking algorithms we looked at, we saw algorithms that found decision points and recursed over all options from that decision point. A **greedy algorithm** can be thought of as a backtracking algorithm where at each decision point "the best" option is already known and thus can be picked without having to recurse over any of the alternative options.

The name "greedy" comes from the fact that the algorithms make decisions based on a single criterion, instead of a global analysis that would take into account the decision's effect on further steps. As we will see, such a backtracking analysis will be unnecessary in the case of greedy algorithms, so it is not greedy in the sense of causing harm for only short-term gain.

Unlike backtracking algorithms, greedy algorithms can't be made for every problem. Not every problem is "solvable" using greedy algorithms. Viewing the finding solution to an optimization problem as a hill climbing problem greedy algorithms can be used for only those hills where at every point taking the steepest step would lead to the peak always.

Greedy algorithms tend to be very efficient and can be implemented in a relatively straightforward fashion. Many a times in O(n) complexity as there would be a single choice at every point. However, most attempts at creating a correct greedy algorithm fail unless a precise proof of the algorithm's correctness is first demonstrated. When a greedy strategy fails to produce optimal results on all inputs, we instead refer to it as a heuristic instead of an algorithm. Heuristics can be useful when speed is more important than exact results (for example, when "good enough" results are sufficient).

Event Scheduling Problem

The first problem we'll look at that can be solved with a greedy algorithm is the event scheduling problem. We are given a set of events that have a start time and finish time, and we need to produce a subset of these events such that no events intersect each other (that is, having overlapping times), and that we have the maximum number of events scheduled as possible.

Here is a formal statement of the problem:

Input: *events*: a set of intervals where is the start time, and is the finish time.
Solution: A subset *S* of *Events*.
Constraint: No events can intersect (start time exclusive). That is, for all intervals where it holds that .
Objective: Maximize the number of scheduled events, i.e. maximize the size of the set *S*.

We first begin with a backtracking solution to the problem:

```
// event-schedule -- schedule as many non-conflicting events as possible
function event-schedule(events array of s[1..n], f[1..n]): set
  if n == 0: return   fi
  if n == 1: return {events[1]} fi
  let event := events[1]
  let S1 := union(event-schedule(events - set of conflicting events), event)
  let S2 := event-schedule(events - {event})
  if S1.size() >= S2.size():
    return S1
  else
    return S2
  fi
end
```

The above algorithm will faithfully find the largest set of non-conflicting events. It brushes aside details of how the set

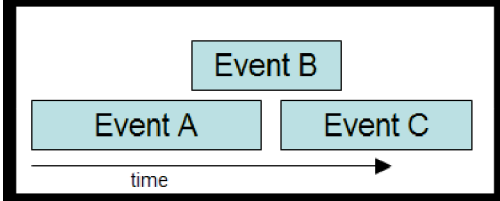
events - set of conflicting events

is computed, but it would require time. Because the algorithm makes two recursive calls on itself, each with an argument of size , and because removing conflicts takes linear time, a recurrence for the time this algorithm takes is:

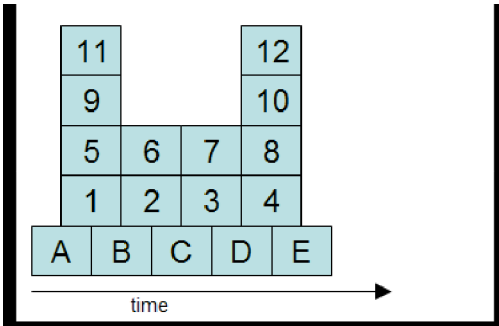
which is .

But suppose instead of picking just the first element in the array we used some other criterion. The aim is to just pick the "right" one so that we wouldn't need two recursive calls. First, let's consider the greedy strategy of picking the shortest events first, until we can add no more events without conflicts. The idea here is that the shortest events would likely interfere less than other events.

There are scenarios were picking the shortest event first produces the optimal result. However, here's a scenario where that strategy is sub-optimal:



Above, the optimal solution is to pick event A and C, instead of just B alone. Perhaps instead of the shortest event we should pick the events that have the least number of conflicts. This strategy seems more direct, but it fails in this scenario:



Above, we can maximize the number of events by picking A, B, C, D, and E. However, the events with the least conflicts are 6, 2 and 7, 3. But picking one of 6, 2 and one of 7, 3 means that we cannot pick B, C and D, which includes three events instead of just two.

= Longest Path solution to critical path scheduling of jobs

Construction with dependency constraints but concurrency can use critical path determination to find minimum time feasible, which is equivalent to a longest path in a directed acyclic graph problem. By using relaxation and breath first search, the shortest path can be the longest path by negating weights(time constraint), finding solution, then restoring the positive weights. (Relaxation is determining the parent with least accumulated weight for each adjacent node being scheduled to be visited)

Dijkstra's Shortest Path Algorithm

With two (high-level, pseudocode) transformations, Dijsktra's algorithm can be derived from the much less efficient backtracking algorithm. The trick here is to prove the transformations maintain correctness, but that's the whole insight into Dijkstra's algorithm anyway. [TODO: important to note the paradox that to solve this problem it's easier to solve a more-general version. That is, shortest path from s to all nodes, not just to t. Worthy of its own colored box.]

To see the workings of Dijkstra's Shortest Path Algorithm, take an example:

There is a start and end node, with 2 paths between them ; one path has cost 30 on first hop, then 10 on last hop to the target node, with total cost 40. Another path cost 10 on first hop, 10 on second hop, and 40 on last hop, with total cost 60.

The start node is given distance zero so it can be at the front of a shortest distance queue, all the other nodes are given infinity or a large number e.g. 32767 .

This makes the start node the first current node in the queue.

With each iteration, the current node is the first node of a shortest path queue. It looks at all nodes adjacent to the current node;

For the case of the start node, in the first path it will find a node of distance 30, and in the second path, an adjacent node of distance 10. The current nodes distance, which is zero at the beginning, is added to distances of the adjacent nodes, and the distances from the start node of each node are updated, so the nodes will be 30+0 = 30 in the 1st path, and 10+0=10 in the 2nd path.

Importantly, also updated is a previous pointer attribute for each node, so each node will point back to the current node, which is the start node for these two nodes.

Each node's priority is updated in the priority queue using the new distance.

That ends one iteration. The current node was removed from the queue before examining its adjacent nodes.

In the next iteration, the front of the queue will be the node in the second path of distance 10, and it has only one adjacent node of distance 10, and that adjacent node will distance will be updated from 32767 to 10 (the current node distance) + 10 (the distance from the current node) = 20.

In the next iteration, the second path node of cost 20 will be examined, and it has one adjacent hop of 40 to the target node, and the target nodes distance is updated from 32767 to 20 + 40 = 60 . The target node has its priority updated.

In the next iteration, the shortest path node will be the first path node of cost 30, and the target node has not been yet removed from the queue. It is also adjacent to the target node, with the total distance cost of 30 + 10 = 40.

Since 40 is less than 60, the previous calculated distance of the target node, the target node distance is updated to 40, and the previous pointer of the target node is updated to the node on the first path.

In the final iteration, the shortest path node is the target node, and the loop exits.

Looking at the previous pointers starting with the target node, a shortest path can be reverse constructed as a list to the start node.

Given the above example, what kind of data structures are needed for the nodes and the algorithm ?

```
# author, copyright under GFDL
class Node :
    def __init__(self, label, distance = 32767 ):
    # a bug in constructor, uses a shared map initializer
        #, adjacency_distance_map = {} ):
      self.label = label

      self.adjacent = {}  # this is an adjacency map, with keys nodes, and values the adjacent distance

      self.distance = distance   # this is the updated distance from the start node, used as the node's priority
      # default distance is 32767

      self.shortest_previous = None  #this the last shortest distance adjacent node

    # the logic is that the last adjacent distance added is recorded, for any distances of the same node added
    def add_adjacent(self, local_distance, node):
```

```
        self.adjacent[node]=local_distance
        print "adjacency to ", self.label, " of ", self.adjacent[node], " to ", \
         node.label

    def get_adjacent(self) :
        return self.adjacent.iteritems()

    def update_shortest( self, node):
    new_distance = node.adjacent[self] + node.distance

    #DEBUG
    print "for node ", node.label, " updating ", self.label, \
        " with distance ", node.distance, \
        " and adjacent distance ", node.adjacent[self]

    updated = False
        # node's adjacency map gives the adjacent distance for this node
        # the new distance for the path to this (self)node is the adjacent distance plus the other node's distance
        if new_distance < self.distance :
                # if it is the shortest distance then record the distance, and make the previous node that node
        self.distance = new_distance
            self.shortest_previous= node
        updated = True
    return updated

MAX_IN_PQ = 100000
class PQ:
        def __init__(self,  sign  = -1 ):
           self.q = [None ] * MAX_IN_PQ # make the array preallocated
           self.sign = sign  # a negative sign is a minimum priority queue
           self.end = 1 # this is the next slot of the array (self.q) to be used,
           self.map = {}

        def  insert( self,  priority, data):
            self.q[self.end] = (priority, data)
            # sift up after insert
            p = self.end
            self.end = self.end + 1
            self.sift_up(p)

        def sift_up(self, p):
            # p is the current node's position
            # q[p][0] is the priority, q[p][1] is the item or node

            # while the parent exists ( p >= 1), and parent's priority is less than the current node's priority
            while  p / 2 != 0 and  self.q[p/2][0]*self.sign  < self.q[p][0]*self.sign:
               # swap the parent and the current node, and make the current node's position the parent's position
               tmp = self.q[p]
               self.q[p] = self.q[p/2]
               self.q[p/2] = tmp
           self.map[self.q[p][1]] = p
               p = p/2

            # this map's the node to the position in the priority queue
            self.map[self.q[p][1]] = p

            return p


        def remove_top(self):

             if  self.end == 1 :
               return (-1, None)

             (priority, node) = self.q[1]
             # put the end of the heap at the top of the heap, and sift it down to adjust the heap
             # after the heap's top has been removed. this takes log2(N) time, where N iis the size of the heap.

             self.q[1] = self.q[self.end-1]
             self.end = self.end - 1

             self.sift_down(1)

             return (priority, node)

        def sift_down(self, p):
            while 1:
              l = p * 2

              # if the left child's position is more than the size of the heap,
              # then left and right children don't exist
              if ( l > self.end) :
                break

              r= l + 1
              # the selected child node should have the greatest priority
              t = l
              if r < self.end and self.q[r][0]*self.sign > self.q[l][0]*self.sign :
                t = r
              print "checking for sift down of ", self.q[p][1].label, self.q[p][0], " vs child ", self.q[t][1].label, self.q[t][0]
              # if the selected child with the greatest priority has a higher priority than the current node
              if self.q[t] [0] * self. sign  >  self.q [p] [0] * self.sign :
                # swap the current node with that child, and update the mapping of the child node to its new position
                tmp = self. q [ t ]
                self. q [ t ] = self.q [ p ]
                self. q [ p ] = tmp
                self.map [ tmp [1 ] ] = p
                p = t
              else: break    # end the swap if the greatest priority child has a lesser priority than the current node

            # after the sift down, update the new position of the current node.
            self.map [ self.q[p][1] ] = p
            return p

        def  update_priority(self, priority, data ) :

             p = self. map[ data ]
         print "priority prior update", p, "for priority", priority, " previous priority", self.q[p][0]
            if p is None :
               return -1

            self.q[p]  = (priority, self.q[p][1])
            p = self.sift_up(p)
```

```
            p = self.sift_down(p)
        print "updated ", self.q[p][1].label, p, "priority now ", self.q[p][0]

            return p

class NoPathToTargetNode ( BaseException):
  pass

def test_1() :
    st =  Node('start', 0)
    p1a =  Node('p1a')
    p1b =  Node('p1b')
    p2a =  Node('p2a')
    p2b =  Node('p2b')
    p2c = Node('p2c')
    p2d = Node('p2d')
    targ =  Node('target')
    st.add_adjacent ( 30, p1a)
    #st.add_adjacent ( 10, p2a)
    st.add_adjacent ( 20, p2a)
    #p1a.add_adjacent(10, targ)
    p1a.add_adjacent(40, targ)
    p1a.add_adjacent(10, p1b)
    p1b.add_adjacent(10, targ)
    # testing alternative
    #p1b.add_adjacent(20, targ)
    p2a.add_adjacent(10, p2b)
    p2b.add_adjacent(5,p2c)
    p2c.add_adjacent(5,p2d)
    #p2d.add_adjacent(5,targ)
    #chooses the alternate path
    p2d.add_adjacent(15,targ)
    pq =  PQ()

    # st.distance is 0, but the other's have default starting distance 32767
    pq.insert( st.distance, st)
    pq.insert( p1a.distance, p1a)
    pq.insert( p2a.distance, p2a)
    pq.insert( p2b.distance, p2b)
    pq.insert(targ.distance, targ)
    pq.insert( p2c.distance, p2c)
    pq.insert( p2d.distance, p2d)

    pq.insert(p1b.distance, p1b)

    node = None

    while  node !=  targ :
     (pr, node ) = pq.remove_top()
     #debug
     print "node ", node.label, " removed from top "
     if  node is None:
              print "target node not in queue"
              raise
     elif pr == 32767:
              print "max distance encountered so no further nodes updated. No path to target node."
              raise NoPathToTargetNode

     # update the distance to the start node using this node's distance to all of the nodes adjacent to it, and update its priority if
     # a shorter distance was found for an adjacent node ( .update_shortest(..) returns true ).
     # this is the greedy part of the dijsktra's algorithm, always greedy for the shortest distance using the priority queue.
     for adj_node, dist in node.get_adjacent():
       #debug
       print "updating adjacency from ", node.label, " to ", adj_node.label
       if adj_node.update_shortest( node ):
       pq.update_priority(  adj_node.distance, adj_node)

     print "node and targ ", node, targ, node <> targ
    print "length of path", targ.distance
    print " shortest path"

    #create a reverse list from the target node, through the shortes path nodes to the start node
    node = targ

    path = []
    while node <> None :
       path.append(node)
       node = node. shortest_previous

    for node in reversed(path):  # new iterator version of list.reverse()
       print node.label

if __name__ == "__main__":
  test_1()
```

Minimum spanning tree

Greedily looking for the minimum weight edges; this could be achieved with sorting edges into a list in ascending weight. Two well known algorithms are Prim's Algorithm and Kruskal's Algorithm. Kruskal selects the next minimum weight edge that has the condition that no cycle is formed in the resulting updated graph. Prim's algorithm selects a minimum edge that has the condition that only one edge is connected to the tree. For both the algorithms, it looks that most work will be done verifying an examined edge fits the primary condition. In Kruskal's, a search and mark technique would have to be done on the candidate edge. This will result in a search of any connected edges already selected, and if a marked edge is encountered, than a cycle has been formed. In Prim's algorithm, the candidate edge would be compared to the list of currently selected edges, which could be keyed on vertex number in a symbol table, and if both end vertexes are found, then the candidate edge is rejected.

Maximum Flow in weighted graphs

In a flow graph, edges have a forward capacity, a direction, and a flow quantity in the direction and less than or equal to the forward capacity. Residual capacity is capacity minus flow in the direction of the edge, and flow in the other direction.

Maxflow in Ford Fulkerson method requires a step to search for a viable path from a source to a sink vertex, with non-zero residual capacities at each step of the path. Then the minimum residual capacity determines the maximum flow for this path. Multiple iterations of searches using BFS can be done (the Edmond-Karp algorithm), until the sink vertex is not marked when the last node is off the queue or stack. All marked nodes in the last iteration are said to be in the minimum cut.

Here are 2 java examples of implementation of Ford Fulkerson method, using BFS. The first uses maps to map vertices to input edges, whilst the second avoids the Collections types Map and List, by counting edges to a vertex and then allocating space for each edges array indexed by vertex number, and by using a primitive list node class to implement the queue for BFS.

For both programs, the input are lines of "vertex_1, vertex_2, capacity", and the output are lines of "vertex_1, vertex_2, capacity, flow", which describe the initial and final flow graph.

```
// copyright GFDL and CC-BY-SA

package test.ff;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;

public class Main {
    public static void main(String[] args) throws IOException {
        System.err.print("Hello World\n");
        final String filename = args[0];
        BufferedReader br = new BufferedReader( new FileReader(filename));
        String line;
        ArrayList<String[]> lines = new ArrayList<>();

        while ((line= br.readLine()) != null) {
            String[] toks = line.split("\\s+");
            if (toks.length == 3)
                lines.add(toks);
            for (String tok : toks) {

                System.out.print(tok);
                System.out.print("-");
            }
            System.out.println("");
        }

        int [][]edges = new int[lines.size()][4];

        // edges, 0 is from-vertex, 1 is to-vertex, 2 is capacity, 3 is flow

        for (int i = 0; i < edges.length; ++i)
            for (int j =0; j < 3; ++j)
                edges[i][j] = Integer.parseInt(lines.get(i)[j]);

        Map<Integer, List<int[]>> edgeMap = new HashMap<>();

        // add both ends into edge map for each edge
        int last = -1;
        for (int i = 0; i < edges.length; ++i)
            for (int j = 0; j < 2; ++j) {
                edgeMap.put(edges[i][j],
                        edgeMap.getOrDefault(edges[i][j],
                                new LinkedList<int[]>()) );
                edgeMap.get(edges[i][j]).add(edges[i]);

                // find the highest numbered vertex, which will be the sink.
                if ( edges[i][j] > last )
                    last = edges[i][j];
            }

        while(true) {

            boolean[] visited = new boolean[edgeMap.size()];

            int[] previous = new int[edgeMap.size()];
            int[][] edgeTo = new int[edgeMap.size()][];

            LinkedList<Integer> q = new LinkedList<>();
            q.addLast(0);
            int v = 0;
            while (!q.isEmpty()) {
                v = q.removeFirst();

                visited[v] = true;
                if (v == last)
                    break;

                int prevQsize = q.size();

                for ( int[] edge: edgeMap.get(v)) {
                    if (v == edge[0] &&
                            !visited[edge[1]]  &&
                            edge[2]-edge[3] > 0)
                        q.addLast(edge[1]);
                    else if( v == edge[1] &&
                            !visited[edge[0]] &&
                            edge[3] > 0 )
                        q.addLast(edge[0]);
                    else
                        continue;
                    edgeTo[q.getLast()]  = edge;
                }

                for (int i = prevQsize; i < q.size(); ++i) {
                    previous[q.get(i)] = v;

                }


            }

            if ( v == last) {
                int a = v;
                int b = v;
                int smallest = Integer.MAX_VALUE;
                while (a != 0) {
                // get the path by following previous,
```

```
                // also find the smallest forward capacity
                    a = previous[b];
                    int[] edge = edgeTo[b];
                    if ( a == edge[0] && edge[2]-edge[3] < smallest)
                        smallest = edge[2]-edge[3];
                    else if (a == edge[1] &&  edge[3] < smallest )
                        smallest = edge[3];
                    b = a;
                }

                // fill the capacity along the path to the smallest
                b = last; a = last;
                while ( a != 0) {
                    a = previous[b];
                    int[] edge = edgeTo[b];
                    if ( a == edge[0] )
                        edge[3] = edge[3] + smallest;
                    else
                        edge[3] = edge[3] - smallest;
                    b = a;
                }

            } else {
                // v != last, so no path found
                // max flow reached
                break;
            }

        }

        for ( int[] edge: edges) {
            for ( int j = 0; j < 4; ++j)
                System.out.printf("%d-",edge[j]);
            System.out.println();
        }
    }
}
```

```
// copyright GFDL and CC-BY-SA
package test.ff2;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;

public class MainFFArray {

    static class Node {
        public Node(int i) {
            v = i;
        }

        int v;
        Node next;
    }

    public static void main(String[] args) throws IOException {
        System.err.print("Hello World\n");
        final String filename = args[0];
        BufferedReader br = new BufferedReader(new FileReader(filename));
        String line;
        ArrayList<String[]> lines = new ArrayList<>();

        while ((line = br.readLine()) != null) {
            String[] toks = line.split("\\s+");
            if (toks.length == 3)
                lines.add(toks);
            for (String tok : toks) {

                System.out.print(tok);
                System.out.print("-");
            }
            System.out.println("");
        }

        int[][] edges = new int[lines.size()][4];

        for (int i = 0; i < edges.length; ++i)
            for (int j = 0; j < 3; ++j)
                edges[i][j] = Integer.parseInt(lines.get(i)[j]);

        int last = 0;
        for (int[] edge : edges) {
            for (int j = 0; j < 2; ++j)
                if (edge[j] > last)
                    last = edge[j];
        }

        int[] ne = new int[last + 1];

        for (int[] edge : edges)
            for (int j = 0; j < 2; ++j)
                ++ne[edge[j]];

        int[][][] edgeFrom = new int[last + 1][][];

        for (int i = 0; i < last + 1; ++i)
            edgeFrom[i] = new int[ne[i]][];

        int[] ie = new int[last + 1];

        for (int[] edge : edges)
            for (int j = 0; j < 2; ++j)
                edgeFrom[edge[j]][ie[edge[j]]++] = edge;

        while (true) {
            Node head = new Node(0);
            Node tail = head;
            int[] previous = new int[last + 1];
```

```
74
75              for (int i = 0; i < last + 1; ++i)
76                  previous[i] = -1;
77
78
79              int[][] pathEdge = new int[last + 1][];
80
81              while (head != null ) {
82                  int v = head.v;
83                  if (v==last)break;
84                  int[][] edgesFrom = edgeFrom[v];
85
86                  for (int[] edge : edgesFrom) {
87                      int nv = -1;
88                      if (edge[0] == v && previous[edge[1]] == -1 && edge[2] - edge[3] > 0)
89                          nv = edge[1];
90                       else if (edge[1] == v && previous[edge[0]] == -1 && edge[3] > 0)
91                           nv = edge[0];
92                       else
93                           continue;
94
95                      Node node = new Node(nv);
96                      previous[nv]=v;
97                      pathEdge[nv]=edge;
98
99                      tail.next = node;
100                     tail = tail.next;
101
102                 }
103
104                 head = head.next;
105
106             }
107
108             if (head == null)
109                 break;
110
111             int v = last;
112
113             int minCapacity = Integer.MAX_VALUE;
114
115             while (v != 0) {
116                 int fv = previous[v];
117                 int[] edge = pathEdge[v];
118                 if (edge[0] == fv && minCapacity > edge[2] - edge[3])
119                     minCapacity = edge[2] - edge[3];
120                 else if (edge[1] == fv && minCapacity > edge[3])
121                     minCapacity = edge[3];
122                 v = fv;
123             }
124
125             v = last;
126             while (v != 0) {
127                 int fv = previous[v];
128                 int[] edge = pathEdge[v];
129                 if (edge[0] == fv)
130                     edge[3] += minCapacity;
131                 else if (edge[1] == fv)
132                     edge[3] -= minCapacity;
133
134                 v = fv;
135             }
136
137         }
138
139         for (int[] edge : edges) {
140             for (int j = 0; j < 4; ++j)
141                 System.out.printf("%d-", edge[j]);
142             System.out.println();
143         }
144
145     }
146
147 }
```

Top, Chapters: 1, 2, 3, 4, 5, 6, 7, 8, 9, A