Greedy Algorithms

CS 4102: Algorithms

Mark Floryan

Overview

Inventory of greedy algorithms

- Coin change
- Knapsack algorithm
- Interval scheduling
- Prim's MST
- Kruskal's MST
- Dijkstra's shortest path
- Huffman Codes

Greedy Method: Overview

- Optimization problems: terminology
  - A solution must meet certain constraints: A solution is *feasible*
    - Example: All edges in solution are in graph, form a simple path
  - Solutions judged on some criteria: *Objective function*
    - Example: Sum of edge weights in path is smallest
  - One (or more) feasible solutions that scores best (by the objective function) is the optimal solution(s)

Greedy Method: Overview

- Greedy strategy:
  - Build solution by stages, adding one item to partial solution found so far
  - At each stage, make locally optimal choice based on the greedy rule (sometimes called the selection function)
    - Locally optimal, i.e. best given what info we have now
  - Irrevocable, a choice can't be un-done
  - Sequence of locally optimal choices leads to globally optimal solution (hopefully)
    - Must prove this for a given problem!
    - Approximation algorithms, heuristics

Proving them correct

- Given a greedy algorithm, how do you show it is optimal?
  - As opposed to other types of algorithms (divide-and-conquer , etc.)
- One common way is to compare the solution given with an optimal solution

Making Change

Everyone Already Knows Many Algorithms!

- Worked retail? You know how to make change!
- Example:
  - My item costs $4.37. I give you a five dollar bill. What do you give me in change?
  - Answer: two quarters, a dime, three pennies
  - Why? How do we figure that out?

Making Change

- The problem:
  - Give back the right amount of change, and…
  - Return the fewest number of coins!
- Inputs: the dollar-amount to return
  - Also, the set of possible coins. (Do we have half-dollars? That affects the answer we give.)
- Output: a set of coins

- Note this problem statement is simply a transformation
  - Given input, generate output with certain properties
  - No statement about how to do it.
- Can you describe the algorithm you use?

A Change Algorithm

1. Consider the largest coin
2. How many go into the amount left?
3. Add that many of that coin to the output
4. Subtract the amount for those coins from the amount left to return
5. If the amount left is zero, done!
6. If not, consider next largest coin, and go back to Step 2

Is this a "good" algorithm?

- What makes an algorithm "good"?
  - Good time *complexity*. (Maybe space complexity.)
  - Better than any other algorithm
  - Easy to understand
- How could we measure how much work an algorithm does?
  - Code it and time it. Issues?
  - Count how many "instructions" it does before implementing it
  - Computer scientists count basic operations, and use a rough measure of this: order class, e.g. O(n lg n)

Evaluating Our Greedy Algorithm

- How much work does it do?
  - Say C is the amount of change, and N is the number of coins in our coin-set
  - Loop at most N times, and inside the loop we do:
    - A division
    - Add something to the output list
    - A subtraction, and a test
  - We say this is O(N), or linear in terms of the size of the coin-set
- Could we do better?
  - Is this an *optimal algorithm*?
  - We need to do a proof somehow to show this

Formal algorithmic description

- *All* algorithms in this course must have the following components:
  - Problem description (1 line max)
  - Inputs
  - Outputs
  - Assumptions
  - Strategy overview
    - 1 or 2 sentences outlining the basic strategy, including the name of the method you are going to use for the algorithm
  - Algorithm description
    - If listed in English (as opposed to pseudo-code), then it should be listed in steps

Change solution (greedy)

- **Problem description:** providing coin change of a given amount in the fewest number of coins
- **Inputs:** the dollar-amount to return. Perhaps the possible set of coins, if it is non-obvious.
- **Output:** a set of coins that obtains the desired amount of change in the fewest number of coins
- **Assumptions:** If the coins are not stated, then they are the standard quarter, dime, nickel, and penny. All inputs are non-negative, and dollar amounts are ignored.
- **Strategy:** a greedy algorithm that uses the largest coins first
- **Description:** Issue the largest coin (quarters) until the amount left is less than the amount of a quarter ($0.25). Repeat with decreasing coin sizes (dimes, nickels, pennies).

Another Change Algorithm

- Give me another way to do this?
- Brute force:
  - Generate all possible combinations of coins that add up to the required amount
  - From these, choose the one with smallest number
- What would you say about this approach?
- There are other ways to solve this problem
  - *Dynamic programming*: build a table of solutions to small subproblems, work your way up

Change solution (brute-force)

- **Problem description:** providing coin change of a given amount in the fewest number of coins
- **Inputs:** the dollar-amount to return. Perhaps the possible set of coins, if it is non-obvious.
- **Output:** a set of coins that obtains the desired amount of change in the fewest number of coins
- **Assumptions:** If the coins are not stated, then they are the standard quarter, dime, nickel, and penny. All inputs are non-negative, and dollar amounts are ignored.
- **Strategy:** a brute-force algorithm that considers every possibility and picks the one with the fewest number of coins
- **Description:** Consider every possible combination of coins that add to the given amount (done via a depth-first search). Return the one with the fewest number of coins.

Algorithm for making change

- This algorithm makes change for an amount A using coins of denominations

$$denom[1] > denom[2] > \cdots > denom[n] = 1.$$

- `Input Parameters: denom, A`
- `Output Parameters: None`

- greedy_coin_change(denom, A)

```
greedy_coin_change(denom, A) {
  i = 1
  while (A > 0) {
     c = A / denom[i]
     println("use " + c + " coins of denomination " + denom[i])
     A = A - c * denom[i]
     i = i + 1
  }
}
```

Making change proof

- Prove that the provided making change algorithm is optimal for denominations 1, 5, and 10

- Via induction, and on board -->

Formal proof

- Formal proof of the change problem
- Algorithm 7.1.1 is what is presented two slides previously

Theorem 7.1.2. *Algorithm 7.1.1 is optimal for denominations 1, 5, and 10.*

Proof. We use induction on A to prove that to make change for an amount A, the output of Algorithm 7.1.1 and the optimal solution are identical. The cases $A = 1, 2, 3, 4, 5, 10$ are readily verified.

The inductive assumption is that to make change for an amount k, where $k < A$, the output of Algorithm 7.1.1 and the optimal solution are identical. Suppose first that $5 < A < 10$. Let *Opt* be an optimal solution. Now *Opt* must use a coin of denomination 5. (If *Opt* does not use a coin of denomination 5, it is restricted to coins of denomination 1. Because $A > 5$, *Opt* must use at least five 1's. But now *Opt* is not optimal because it could trade in five coins of denomination 1 for one of denomination 5.) Now *Opt* with one coin of denomination 5 removed is optimal for $A - 5$. (If *Opt* with one coin of denomination 5 removed is not optimal for $A - 5$, there is another solution for $A - 5$ that uses fewer coins. Adding a coin of denomination 5 to the solution to the $A - 5$ problem produces a solution for A using fewer coins than *Opt*, which is impossible.) By the inductive assumption, the output of Algorithm 7.1.1 for $A - 5$ and *Opt* with one coin of denomination 5 removed are identical. Adding a coin of denomination 5 to the output of Algorithm 7.1.1 for $A - 5$ yields the output of Algorithm 7.1.1 for A. Thus, the output of Algorithm 7.1.1 and the optimal solution are identical for $5 < A < 10$.

The argument is similar for the case $A > 10$, so we omit some details. Suppose that $A > 10$. Let *Opt* be an optimal solution. Now *Opt* must use a coin of denomination 10. Then *Opt* with one coin of denomination 10 removed is optimal for $A - 10$. By the inductive assumption, the output of Algorithm 7.1.1 for $A - 10$ and *Opt* with one coin of denomination 10 removed are identical. Adding a coin of denomination 10 to the output of Algorithm 7.1.1 for $A - 10$ yields the output of Algorithm 7.1.1 for A. Thus, the output of Algorithm 7.1.1 and the optimal solution are identical for $A > 10$. The inductive step is complete. ■

How would a failed proof work?

- Prove that the provided making change algorithm is optimal for denominations 1, 6, and 10
- Via induction, and on board -->

Knapsack Algorithm

Knapsack Problems

- Motivated by a theoretical burglary scenario (realistically motivated by other similar problems)
- A thief breaks into a house and must gather as many precious items as possible.
- BUT…he cannot gather items, the total weight of which, exceeds the capacity of his knapsack.

Knapsack Problems

- Inputs:
  - n items, each with a weight w_i and a value v_i
  - capacity of the knapsack, C
- Output:
  - Fractions for each of the n items, x_i
    - Or…the actual weights of each item taken
  - Chosen to maximize total profit but not to exceed knapsack capacity

Two Types of Knapsack Problem

- 0/1 knapsack problem (or discrete knapsack)
  - Each item is discrete. Must choose all of it or none of it. So each x_i is 0 or 1
  - Greedy approach does not produce optimal solutions
  - But another approach, dynamic programming, does

- Continuous knapsack problem
  - Can pick up fractions of each item
  - The correct selection function yields a greedy algorithm that produces optimal results

Greedy Rule for Knapsack?

- Build up a partial solution by choosing x_i for one item until knapsack is full (or no more items). Which item to choose?
- There are several choices. Pick one and try on this:
  - n = 3, C = 20
  - weights = (18, 15, 10)
  - values = (25, 24, 15)
- What answer do you get?
- The optimal answer is: (0, 1, 0.5), total=31.5
  - Can you verify this?

Possible Greedy Rules for Knapsack

- Build up a partial solution by choosing x_i for one item until knapsack is full (or no more items). Which item to choose?
- Maybe this: take as much as possible of the remaining item that has largest value, v_i
- Or maybe this: take as much as possible of the remaining items that has smallest weight, w_i
- Neither of these produce optimal values! The one that does "combines" these two approaches.
  - Use ratio of profit-to-weight

Example Knapsack Problem

- For this example:
  - n = 3, C = 20
  - weights = (18, 15, 10)
  - values = (25, 24, 15)
- Ratios = (25/18, 24/15, 15/10)
  = (1.39, 1.6, 1.5)
- The optimal answer is: (0, 1, 0.5)

Continuous knapsack algorithm

```
continuous_knapsack(a, C)
    n = a.last
    for i = 1 to n
            ratio[i] = a[i].p / a[i].w
    sort (a,ratio)
    weight = 0
    i = 1
    while ( i ≤ n && weight < C )
            if ( weight + a[i].w ≤ C )
                    println ("select all of object " + a[i].id)
                    weight = weight + a[i].w
            else
                    r = (C – weight) / a[i].w
                    println ("select " + r + " of object " + a[i].id)
                    weight = C
            i++
```

- a is an array containing the items to be put into the knapsack. Each element has the following fields:
  - p: the profit for that item
  - w: the weight for that item
  - id: the identifier for that item
- C is the capacity of the knapsack
- What is the running time?

How do we know it's correct?

- Proof time!!!

- On board -->

How do we know it's correct?

- Proof time!!!

16.2-1 We want to show that the fractional knapsack problem has the property that a choice that yields a local optimum also gives a global optimum.

Let I be the following instance of the knapsack problem: Let n be the number of items, let v_i be the value of the i'th item, let w_i be the weight of the i'th item and let W be the capacity. Assume the items have been ordered in increasing order by v_i/w_i and that $W \geq w_n$. Let $s = (s_1, s_2, \ldots, s_n)$ be a solution. The greedy algorithm works by assigning $s_n = \min(w_n, W)$, and then continuing by solving the subproblem $I' = (n-1, \{v_1, v_2, \ldots, v_{n-1}\}, \{w_1, w_2, \ldots, w_{n-1}\}, W - w_n)$ until it either reaches the state $W = 0$ or $n = 0$.

We need to show that this strategy always gives an optimal solution. We prove this by contradiction. Suppose the optimal solution to I is $s_1, s_2, \ldots, s_n$, where $s_n < \min(w_n, W)$. Let i be the smallest number such that $s_i > 0$. By decreasing s_i to $\max(0, W - w_n)$ and increasing s_n by the same amount, we get a better solution. Since this a contradiction the assumption must be false. Hence the problem has the greedy-choice property.

Interval Selection

Activity-Selection Problem

- Problem: You and your classmates go on Semester at Sea
  - Many exciting activities each morning
  - Each starting and ending at different times
  - Maximize your "education" by doing as many as possible. (They're all equally good!)
- Welcome to the *activity selection problem*
  - Also called *interval scheduling*

The Activities!

Id	Start	End	Activity
1	9:00	10:45	Fractals, Recursion and Crayolas
2	9:15	10:15	Tropical Drink Engineering with Prof. Bloomfield
3	9:30	12:30	Managing Keyboard Fatigue with Swedish Massage
4	9:45	10:30	Applied ChemE: Suntan Oil or Lotion?
5	9:45	11:15	Optimization, Greedy Algorithms, and the Buffet Line
6	10:15	11:00	Hydrodynamics and Surfing
7	10:15	11:30	Computational Genetics and Infectious Diseases
8	10:30	11:45	Turing Award Speech Karaoke
9	11:00	12:00	Pool Tanning for Engineers
10	11:00	12:15	Mechanics, Dynamics and Shuffleboard Physics
11	12:00	12:45	Discrete Math Applications in Gambling

Generalizing Start, End

Id	Start	End	Len	Activity
1	0	6	7	Fractals, Recursion and Crayolas
2	1	4	4	Tropical Drink Engineering with Prof. Bloomfield
3	2	13	12	Managing Keyboard Fatigue with Swedish Massage
4	3	5	3	Applied ChemE: Suntan Oil or Lotion?
5	3	8	6	Optimization, Greedy Algorithms, and the Buffet Line
6	5	7	3	Hydrodynamics and Surfing
7	5	9	5	Computational Genetics and Infectious Diseases
8	6	10	5	Turing Award Speech Karaoke
9	8	11	4	Pool Tanning for Engineers
10	8	12	5	Mechanics, Dynamics and Shuffleboard Physics
11	12	14	3	Discrete Math Applications in Gambling

Greedy Approach

1. Select a first item.
2. Eliminate items that are incompatible with that item. (i.e. they overlap.)
3. Apply the *greedy rule* (AKA *selection function*) to pick the next item.
4. Go to Step 2

What is a good greedy rule for selecting the next item?

Some Possibilities

- Pick the next compatible one that starts earliest
- Pick the shortest one
- Pick the one that has the least conflicts (i.e. overlaps)

- Do any of these work? Counter-examples?

Activity-Selection

- Formally:
  - Given a set S of n activities
    - s_i = start time of activity i
    - f_i = finish time of activity i
  - Find max-size subset A of compatible activities

  - Assume (wlog) that $f_1 \leq f_2 \leq \ldots \leq f_n$

Activity Selection: A Greedy Algorithm

- So actual algorithm is simple:
  - Sort the activities by finish time
  - Schedule the first activity
  - Then schedule the next activity in sorted list which starts after previous activity finishes
  - Repeat until no more activities
- Intuition is even more simple:
  - Always pick next activity that finishes earliest

Activity Selection: Optimal Substructure

- **Let k be the minimum activity in A (i.e., the one with the earliest finish time). Then A - $\{k\}$ is an optimal solution to $S' = \{i \in S: s_i \geq f_k\}$**
  - In words: once activity #1 is selected, the problem reduces to finding an optimal solution for activity-selection over activities in S **compatible** with #1
  - Proof: if we could find optimal solution B' to S' with $|B'| > |A - \{k\}|$,
    - Then B' U $\{k\}$ is compatible
    - And $|B' \cup \{k\}| > |A|$, which is a contradiction

Back to Semester at Sea…

Id	Start	End	Len	Activity
2	1	4	4	Tropical Drink Engineering with Prof. Bloomfield
4	3	5	3	Applied ChemE: Suntan Oil or Lotion?
1	0	6	7	Fractals, Recursion and Crayolas
6	5	7	3	Hydrodynamics and Surfing
5	3	8	6	Optimization, Greedy Algorithms, and the Buffet Line
7	5	9	5	Computational Genetics and Infectious Diseases
8	6	10	5	Turing Award Speech Karaoke
9	8	11	4	Pool Tanning for Engineers
10	8	12	5	Mechanics, Dynamics and Shuffleboard Physics
3	2	13	12	Managing Keyboard Fatigue with Swedish Massage
11	12	14	3	Discrete Math Applications in Gambling

Solution: 2, 6, 9, 11

Visualizing these Activities

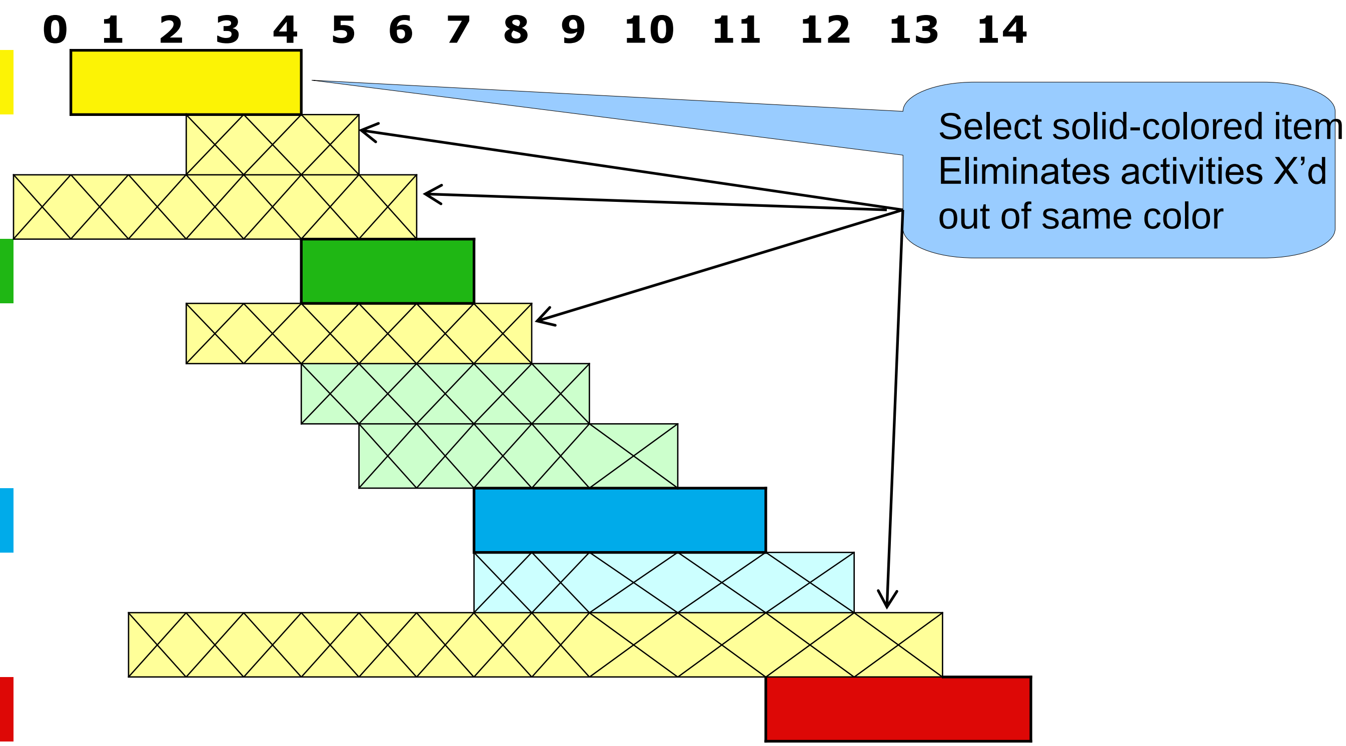
ID	Start	End		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	6																
2	1	4																
3	2	13																
4	3	5																
5	3	8																
6	5	7																
7	5	9																
8	6	10																
9	8	11																
10	8	12																
11	12	14																

Visualizing these Activities

ID	Start	End		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	6																
2	1	4																
3	2	13																
4	3	5																
5	3	8																
6	5	7																
7	5	9																
8	6	10																
9	8	11																
10	8	12																
11	12	14																

Sorted, Then Showing Selection and Incompatibilities

ID	Start	End
2	**1**	**4**
4	3	5
1	0	6
6	**5**	**7**
5	3	8
7	5	9
8	6	10
9	**8**	**11**
10	8	12
3	2	13
11	**12**	**14**



Interval selection algorithm

```
greedy-interval (s, f)
  n = s.length
  A = {a1}
  k = 1
  for m = 2 to n
      if s[m] ≥ f[k]
          A = A U {am}
          k = m
  return A
```

- s is an array of the intervals' start times
- f is an array of the intervals' finish times
- A is the array of the intervals to schedule
- How long does this take?

The proof…

- On board -->
- Show that the greedy algorithm stays ahead!

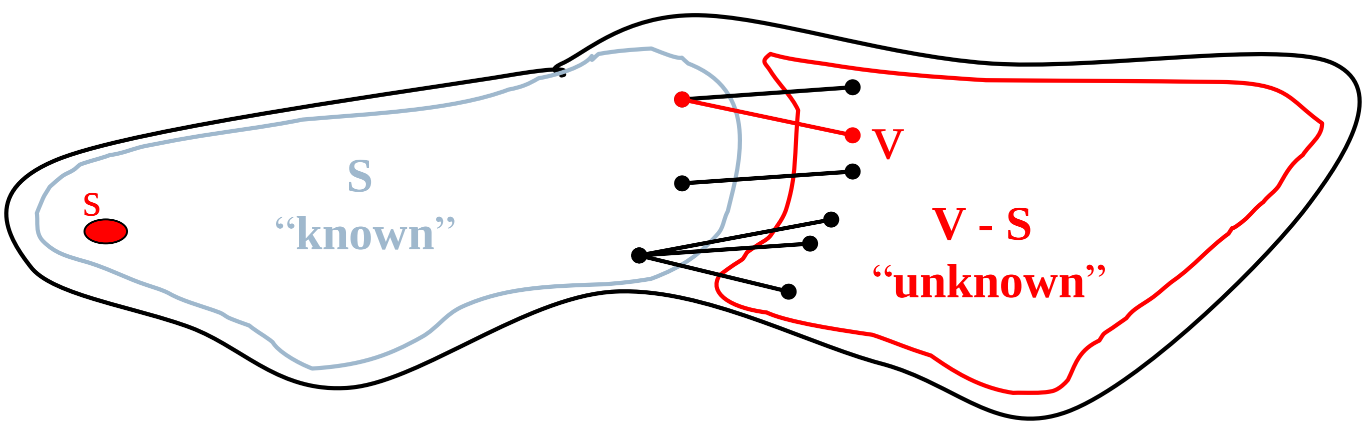
Dijkstra's Shortest Path

Weighted Shortest Path

- no negative weight edges.
- **Dijkstra's algorithm**: uses similar ideas as the unweighted case.

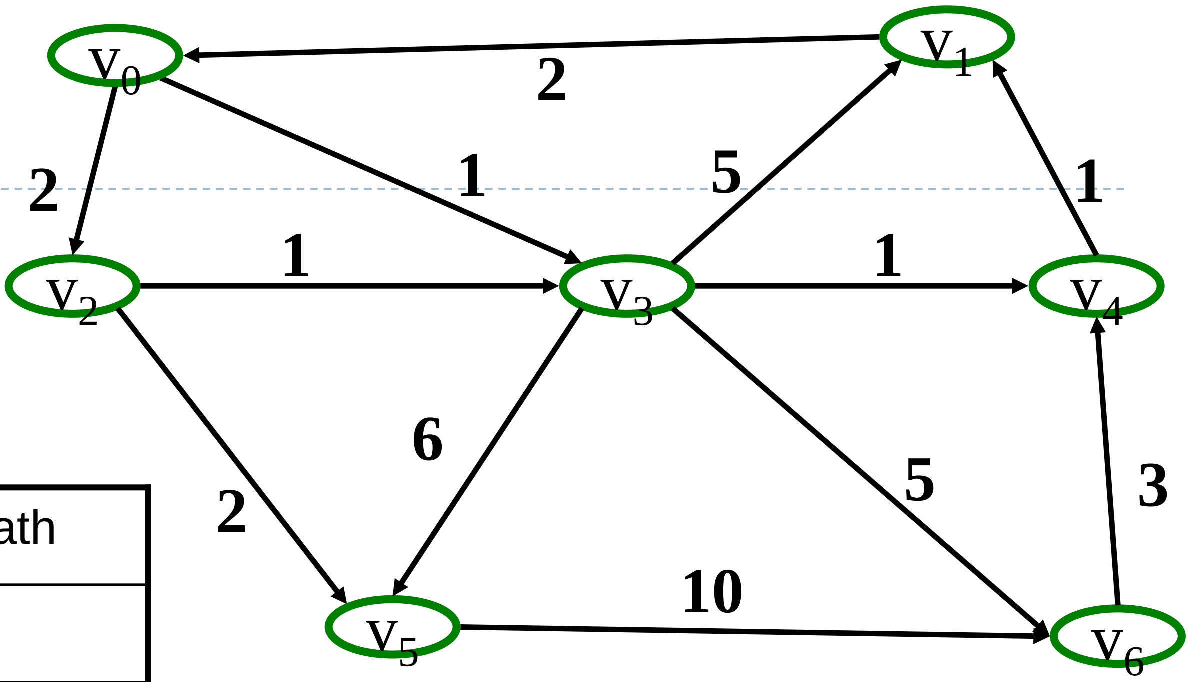
Greedy algorithms:

do what seems to be best at every decision point.



Dijkstra's algorithm

- Initialize each vertex's distance as infinity
- Start at a given vertex *s*
  - Update *s*'s distance to be 0
- Repeat
  - Pick the next unknown vertex with the shortest distance to be the next *v*
    - If no more vertices are unknown, terminate loop
  - Mark *v* as known
  - For each edge from *v* to adjacent unknown vertices *w*
    - If the total distance to *w* is less than the current distance to *w*
      - Update *w*'s distance and the path to *w*



V	Known	Dist	path
v0			
v1			
v2			
v3			
v4			
v5			
v6			

```
void Graph::dijkstra(Vertex s){
  Vertex v,w;
  s.dist = 0;

  while (there exist unknown vertices, find the
          unknown v with the smallest distance)
    v.known = true;

    for each w adjacent to v
      if (!w.known)
        if (v.dist + Cost_VW < w.dist){
          w.dist = v.dist + Cost_VW;
          w.path = v;
        }
  }
}
```

Analysis

- How long does it take to find the smallest unknown distance?
  - simple scan using an array: O(v)
- Total running time:
  - Using a simple scan: $O(v^2+e) = O(v^2)$
- Optimizations?
  - Use adjacency graphs and heaps
  - Assuming that the graph is connected (i.e. e > v-1), then the running time decreases to O(e + v log v)
  - We can simplify this to O(e log v)
  -

Negative Cost Edges?

- Perhaps the graph weights are the amount of fuel expended
  - Positive means fuel was used
  - And passing by a fuel station is a refueling, which is a negative cost edge
- Dijkstra's algorithm does not work for negative cost edges
  - Others do, but are much less efficient
- What about negative cost cycles?

Dijkstra's Shortest Path Algorithm

- Identical *in structure* to Prim's MST algorithm
  - Of course it solves a different problem!
  - Same time complexity
- Additional input parameter(s)
  - Start node v
  - Destination node w (if needed)
- Different output: a path from v to w and a cost (or sets of paths and costs)
  - The tree is the sets of shortest paths to nodes
- Different greedy strategy:
  - Store shortest paths to fringe-nodes in priority queue
  - Store path-distance to node, not just the one edge-weight

Reminder: Prim's Algorithm

```
MST-Prim(G, wt)
 init PQ to be empty;
 PQ.Insert(s, wt=0);
 parent[s] = NULL;
 while (PQ not empty){
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        PQ.Insert(w, wt(v,w));
        parent[w] = v;
     }
     else if (w is fringe && wt[v,w] < fringeWt(w)){
        PQ.decreaseKey(w, wt[v,w]);
        parent[w] = v;
     }
```

Dijkstra' Algorithm

```
dijkstra(G, wt, s)
 init PQ to be empty;
 PQ.Insert(s, dist=0);
 parent[s] = NULL; dist[s] = 0;
 while (PQ not empty)
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        dist[w] = dist[v] + wt(v,w)
        PQ.Insert(w, dist[w] );
        parent[w] = v;
     }
     else if (w is fringe && dist[v] + wt(v,w) <
  dist[w] ) {
        dist[w] = dist[v] + wt(v,w)
        PQ.decreaseKey(w, dist[w]);
        parent[w] = v;
     }
```

Notes on Dijkstra's Algorithm

- Use dist[] to store distances from start to any fringe or tree node
- Store and calculate using distances instead of edge-weights (like in Kruskal's MST)
- What's the output?
  - Tree captured in the parent[] array
  - Shortest distance to each node in dist[] array
  - Trace shortest path in reverse by using parent[] to move from target back to start node, s

```
dijkstra(adj, start, parent) {
   n = adj.last
   for i = 1 to n { key[i] = ∞}  // key is a local array
   key[start] = 0;   predecessor[start] = 0
   // the following statement initializes the
   // container h to the values in the array key
   h.init(key,n)
   for i = 1 to n {
          v = h.min_weight_index()
          min_cost = h.keyval(v)
          v = h.del()
                ref = adj[v]
                while (ref != null) {
                     w = ref.ver
                     if (h.isin(w) && min_cost + ref.weight < h.keyval(w)) {
                         predecessor[w] = v
                         h.decrease(w, min_cost+ref.weight)
                     } // end if
                     ref = ref.next
                } // end while
   } // end for
}
```

Correctness of These Greedy Algorithms

- Recall that the greedy approach may or may not guarantee an optimal result
- Do these produce optimal solutions?
  - The min weight spanning tree? Kruskal's, Prim's
  - The shortest path from s? Dijkstra's
- Answer: Yes, they do.

Proof of Dijkstra's algorithm

- Via induction and contradiction

- On board -->

Conclusion

Greedy algorithm summary

- Algorithms we saw:
  - Coin change
  - Knapsack algorithm
  - Interval scheduling
  - Prim’s MST
  - Kruskal’s MST
  - Dijkstra’s shortest path
  - Huffman Codes

Greedy Method: Overview

- Optimization problems: terminology
  - Solutions judged on some criteria: *Objective function*
    - Example: Sum of edge weights in path is smallest
  - A solution must meet certain constraints: A solution is *feasible*
    - Example: All edges in solution are in graph, form a simple path
  - One (or more) feasible solutions that scores highest (by the objective function) is the optimal solution(s)

Greedy Method: Overview

- Greedy strategy:
  - Build solution by stages, adding one item to partial solution found so far
  - At each stage, make locally optimal choice based on the greedy rule (sometimes called the selection function)
    - Locally optimal, i.e. best given what info we have now
  - Irrevocable, a choice can't be un-done
  - Sequence of locally optimal choices leads to globally optimal solution (hopefully)
    - Must prove this for a given problem!
    - Approximation algorithms, heuristics

Proving them correct

- Given a greedy algorithm, how do you show it is optimal?
  - As opposed to other types of algorithms (divide-and-conquer , etc.)
- One way is to compare the solution given with an optimal solution
- Another way is through induction

Proving a greedy algorithm is correct

1. Show that it fulfills the greedy-choice property
    - Does making a greedy choice at any arbitrary point yield an optimal solution?
    - Consider an optimal solution to a sub-problem
        - Show that making the greedy choice will yield an optimal solution to the overall problem
2. Show that it has optimal sub-structure
    - Show that a solution to a problem contains optimal solutions to sub-problems

- Or use a general proof that compares !

Warning!

- Proving that an algorithm makes a *greedy choice* at each stage is **NOT** the same as showing that the algorithm has the *greedy choice property*
  - The first is a property of the algorithm designed
  - The second shows that making the greedy choice **will** yield an optimal solution to the overall problem