CS 231: Algorithmic Problem Solving

Naomi Nishimura

Module 3

Date of this version: March 25, 2020

WARNING: Drafts of slides are made available prior to lecture for your convenience. After lecture, slides will be updated to reflect material taught. Check the date on this page to make sure you have the correct, updated version.

WARNING: Slides do not include all class material; if you have missed a lecture, make sure to find out from a classmate what material was presented verbally or on the board.

Solving TSP more quickly

TRAVELING SALESPERSON PROBLEM (TSP)

Input: A complete graph with a weight on each edge

Output: A tour of minimum weight

Making local decisions

For optimization problems, the optimal solution can be viewed as the **global optimum** (maximum or minimum), with the measure function giving the largest or smallest value over all feasible solutions.

We attempt to build up a feasible solution step by step, at each point making a **local** choice, based on the information we have about the current partial solution.

For this course, we can consider an algorithm to be greedy even if the local choice doesn't seem to be choosing the maximum or minimum option - that is, the local choice could possibly be arbitrary.

Paradigm: Greedy algorithms

Recipe for a greedy algorithm:

1. Build up the solution step by step.
2. Choose an option that is best at the moment.
3. Augment the solution using the chosen option.
4. Define options for the next step.
5. Repeat the process without ever undoing a decision.

Notes:

- "Best at the moment" can be a largest or smallest value, or even an arbitrary one if all seem equally good.
- Arbitrary and random are not the same.

Greedy algorithm for TSP

Recipe for a greedy algorithm:

1. Build up the solution step by step. *Starting at an arbitrary vertex, add one vertex at a time.*
2. Choose an option that is best at the moment. *From the current (most recently added) vertex, choose an unvisited vertex that can be reached using a lowest-cost edge.*
3. Augment the solution using the chosen option. *Add the chosen vertex to the tour so far.*
4. Define options for the next step. *Remove the chosen vertex from the unvisited vertices.*
5. Repeat the process without ever undoing a decision. *Never remove or move a vertex once it has been added to the tour so far.*

Pseudocode for greedy TSP

GREEDY TSP(*G*)
INPUT: *An input G*
OUTPUT: *A tour*

```
V ← An arbitrary vertex in G (e.g. lowest-numbered index)
Current ← V
Solution ← [V]
Unvisited ← All other vertices in G
while Unvisited is not empty
    Choose a vertex W in Unvisited with smallest weight edge to V
    Current ← W
    Append W to Solution
    Remove W from Unvisited
return Solution
```

Greedy algorithm pseudocode template

```
GREEDY(I)
INPUT:   An input I
OUTPUT: A feasible solution to I
Options ← preprocessed data from I
Solution ← partial solution
while Options is not empty
    select an item x from Options
    add x and possibly other items to Solution
    remove x and possibly other items from Options
return Solution
```

Note: A greedy algorithm is usually easy to write and has relatively low worst-case running time.

Running time of greedy algorithms

The cost of a greedy algorithm is typically $O(A+BC)$, where:

- A is the cost of preprocessing the input,
- B is the number of steps, and
- C is the cost of taking a step.

We view **preprocessing** as steps taken either before an algorithm is run or before data is processed within an algorithm, such as in a loop.

Typical types of preprocessing we will use include determining information about data items and putting data items in order based on that information.

Preprocessing is often used in a greedy algorithm, where data items are ordered by what is considered to be the best choice.

Analysis of running time of TSP

For TSP, edges might be preprocessed to be stored in order by nondecreasing weight, perhaps stored to allow easy access (details in CS 234).

Determining the cost:

- $A =$ cost of preprocessing is dominated by the cost of the rest of the algorithm
- $B =$ number of steps is the number of times we can choose a new vertex ($\Theta(n)$ iterations)
- $C =$ cost of taking a step is the cost of selection and updating ($\Theta(n)$ time, details omitted)
- Total time $O(A + BC)$ is in $\Theta(n^2)$

Proving correctness of a greedy algorithm

A correct greedy algorithm satisfies two properties:

- **Optimal Substructure Property**: An algorithm relies on **optimal substructure** when the optimal solution to an instance of a problem can be formed from optimal solutions to one or more smaller instances formed from the original instance.
- **Greedy Choice Property**: There is an optimal solution consistent with the greedy choices made by the algorithm.

Note: We will typically omit formal proofs of the Greedy Choice Property from this course.

Assessing a greedy algorithm

Recipe for assessing an algorithm:

1. Determine the (worst-case) running time.
2. Ensure that the algorithm produces the correct output for any instance.

For greedy algorithms:

1. Typically, the cost is dominated by the product of the number of iterations of the loop and the cost of the body of the loop, where the solution is built up by one step at each iteration.
2. Correctness is proved by showing that the Optimal Substructure Property and the Greedy Choice Property both hold. In this course, we will rarely prove the Greedy Choice Property.

Proving a greedy algorithm is not correct

To prove that an algorithm is correct, we need to prove a statement of the form "On any instance, the algorithm produces the correct output."

To prove that an algorithm is **not** correct, we need to **disprove** such a statement.

We can disprove a statement by showing that the **negation** of the statement is true. (Please see the math guide for more details.)

The negation of the statement is of the form "There exists an instance on which the algorithm produces an output that is not correct."

We can view such an instance as a **counterexample** that can be used to disprove the original statement.

In addition to providing the counterexample, we need to demonstrate that the algorithm produces an incorrect output for that instance.

Proof by (counter)example

A statement about the **existence** of an element of a set that satisfies certain properties is often proved by demonstrating that such an element exists.

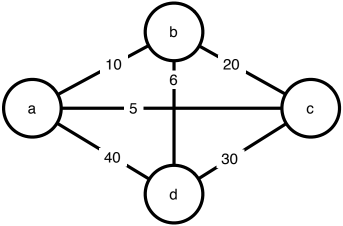
Statement: There exists a computer science course with a course number starting with an even digit.

Proof: CS 231 starts with 2, which is an even digit.

The proof consists of:

- A single example
- Justification that the example satisfies all required properties

Analysis of correctness of Greedy TSP



For our proof, we provide the graph as a counterexample and then show that the algorithm does not result in the optimal tour.
Starting at vertex a results in a, c, b, d, a for a total cost of $5 + 20 + 6 + 40 = 71$.
Starting at vertex b results in b, d, c, a, b for a total cost of $6 + 30 + 5 + 10 = 51$.

Case study: Scheduling activities

You have collected information about all the activities that are possible tomorrow, including when each one starts and each one ends. You've decided that you're not willing to arrive late or leave early, but still want to fit in as many activities as possible. What schedule should you choose?

Scheduling activities

Input: A set of n activities, where activity i has start time s_i and finish time f_i

Output: A schedule that has the maximum number of nonoverlapping activities, that is, for each pair chosen $f_i \leq s_j$ or $f_j \leq s_i$

Aside

There are many variants of this problem: maximize the time spent on activities, allow parts of activities to be scheduled, and so on.

Designing a greedy algorithm for scheduling activities

Recipe for a greedy algorithm:

1. Build up the solution step by step. *Add one activity at a time.*
2. Choose an option that is best at the moment. *Choose an activity (using some criterion to be decided.)*
3. Augment the solution using the chosen option. *Add the activity to the schedule so far.*
4. Define options for the next step. *Remove overlapping activities.*
5. Repeat the process without ever undoing a decision. *Never remove an activity from the schedule so far.*

Choosing a criterion

Idea: If you choose short activities, you can fit in more of them.

Step 2: Choose the shortest remaining activity.

$A \leftarrow$ *activities sorted by nondecreasing length*
$S \leftarrow \emptyset$
while A *is not empty*
    $x \leftarrow$ *any activity in* A *with smallest length*
    $S \leftarrow S \cup \{x\}$
    $A \leftarrow A \setminus \{\textit{activities that overlap } x\}$

Analysis of running time

We assume that sorting can be executed in time $\Theta(n \log n)$ in the worst case (to be discussed later in the course).

All overlapping activities can be removed by checking each one in turn. Faster execution time may be possible depending on the way the data is stored (consider CS 234 for discussion of such ideas).

Determining the cost:

- $A = \Theta(n \log n)$ is the cost of preprocessing the input and initializing S
- $B = O(n)$ is the number of steps (at most the number of activities), and
- $C = O(n)$ is the cost of taking a step
- Total time: $O(A + BC)$ is in $O(n^2)$

Choosing another criterion

Idea: If you choose activities that finish soon, you'll have more time to fit in others.

Step 2: Choose the remaining activity that finishes first.

$A \leftarrow$ *activities sorted by nondecreasing finishing time*
$S \leftarrow \emptyset$
while A *is not empty*
    $x \leftarrow$ *any activity in* A *with smallest finishing time*
    $S \leftarrow S \cup \{x\}$
    $A \leftarrow A \setminus \{$*activities that overlap* $x\}$

Proof by contradiction

We use the following reasoning to show that A is true:

- We first suppose instead that A is false.
- We then use the assumption that A is false to lead to a contradiction, such as a statement that something is both true and false, or that something known to be true (false) is false (true).
- Since a contradiction cannot hold, we can conclude that our assumption was incorrect, and hence A is in fact true.

Please see the math guide for more details.

Proof of optimal substructure

We will use proof by contradiction to establish optimal substructure.

A typical proof will have the following form:

- We wish to prove a statement A about optimal substructure.
- We start with an optimal solution O for the original instance.
- Using the assumption that A is false, we find a new solution S that is better than O.
- The fact that S is better than O contradicts the statement that O is optimal.
- We can then conclude that our assumption was incorrect, and hence A is in fact true.

Recipe for optimal substructure

1. Using the optimal solution O for an arbitrary instance I, construct one or more smaller instances.
2. Decompose O into pieces, one for each smaller instance.
3. Show that if any piece O' of O is not an optimal solution for a smaller instance I' of I, then O is not an optimal solution for I.

Note: If we can show that something holds for an arbitrary instance (that is, by not using any special property of the instance), then we have shown that it holds for every instance.

Proving optimal substructure of SCHEDULING ACTIVITIES

1. Using the optimal solution O for an arbitrary instance I, construct one or more smaller instances. *Form I' by removing the first activity f from O as well as any activity in I that overlaps f.*
2. Decompose O into pieces, one for each smaller instance. *Form O' by removing f from O.*
3. Show that if any piece O' of O is not an optimal solution for a smaller instance I' of I, then O is not an optimal solution for I. *If there is a schedule S' for I' that has more activities than O', then we can form a schedule S for I consisting of I' and f. This is possible since no activities in I' overlap with f. Since S' has more activities than O', S has more activities than O.*

Informal discussion of the greedy choice property

$A = \{a_1, \ldots, a_j\}$ is the solution formed by our algorithm
$O = \{o_1, \ldots, o_k\}$ is an optimal solution

To show:

- Argue that $f(a_i) \leq f(o_i)$ for all values $1 \leq i \leq j$.
- Argue that as a consequence, A must be optimal.

Sketch of arguments:

- This follows from the way items are selected by the algorithm.
- If A is not optimal, then another item could have been added to the schedule. However, O's last item finished no sooner than A's last item.

Note: We will not require formal proofs of the greedy choice property in this course.

Implementing the algorithm

- Represent activities
- Sort activities by finish time
- Check for overlaps
- Form the schedule

Representing activities

```
class Activity:
    def __init__(self, name, start, end):
        self.name = name
        self.start = start
        self.end = end
```

(Code available on the course website as `module3activity.py` with testing file `module3activityuse.py`.)

Sorting activities by finish time

Sorting a Python list:

- `a_list.sort()` mutates `a_list`
- `sorted(a_list)` produces a new list
- `reverse` can be used to sort in nonincreasing order
- `key` can be used to specify how to sort

Example:

```
sorted(a_list, key=my_func, reverse=True)
```

Checking for overlaps

The following method is part of the class definition for `Activity`:

```
def overlap(self, other):
    if self.start <= other.start and
        self.end > other.start:
        return True
    elif other.start <= self.start and
        other.end > self.start:
        return True
    else:
        return False
```

Forming the schedule

Use list operations to form the schedule.

Try writing the entire algorithm as an exercise.

(Code available on the course website as `module3finishfirst.py`, including an optional modification using `filter` and `lambda`.)

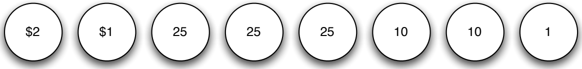
Making change

Making change

Input: A set of integer coin values $c_1, c_2, \ldots, c_k$ and an integer x

Output: Numbers of coins of each denomination such that the total number of coins is the smallest possible and the total value is x

Example: Make change of \$3.96 using coins with denominations 1, 5, 10, and 25 cents, one dollar, and two dollars.



Greedy algorithm for making change

Recipe for a greedy algorithm:

1. Build up the solution step by step. *Add one coin at a time.*
2. Choose an option that is best at the moment. *Choose the maximum denomination coin less than or equal to the remaining change to be made.*
3. Augment the solution using the chosen option. *Add the chosen coin to the solution so far.*
4. Define options for the next step. *Reduce the remaining change by the value of the chosen coin.*
5. Repeat the process without ever undoing a decision. *Never remove a coin from the solution so far.*

Proving optimal substructure

Making change

Input: A set of integer coin values $c_1, c_2, \ldots, c_k$ and an integer x

Output: Numbers of coins of each denomination such that the total number of coins is the smallest possible and the total value is x

1. Using the optimal solution O for an arbitrary instance I, construct one or more smaller instances. *Given an optimal solution O for instance I with total value x, choose an arbitrary coin c_i in O, and form an instance I' in which the total value is $x - c_i$*
2. Decompose O into pieces, one for each smaller instance. *O' is formed by removing a copy of c_i from O*
3. Show that if any piece O' of O is not an optimal solution for a smaller instance I' of I, then O is not an optimal solution for I. *If there is a better solution B for I', then it can be combined with c_i to form a better solution than O for I.*

Greedy sorting

SORTING

Input: A set of numbers

Output: The numbers in order from smallest to largest

Recipe for a greedy algorithm:

1. Build up the solution step by step. *Add one number at a time to the final sorted sequence.*
2. Choose an option that is best at the moment. *Choose the minimum number among the remaining numbers.* **Use what you already know** for this.
3. Augment the solution using the chosen option. *Add the number to the end of the sequence so far.*
4. Define options for the next step. *Remove the chosen number from the remaining options.*
5. Repeat the process without ever undoing a decision. *Never change the order of a number once placed in the sequence.*

Sorting in place

One phase per element:

- At the beginning of phase i, items in the first $i-1$ positions are in order.
- In phase i, search for the minimum element in the range from position i through position n.
- Swap elements so that the minimum element is now in position i.
- At the end of phase i, items in the first i positions are in order.

Selection sort

INPUT

4	2	6	3	1	5

```
def select_sort(alist):
    length = len(alist)
    for phase in range(length):
        smallest_so_far = phase
        for choice in range(phase+1, length):
            if alist[choice] < alist[smallest_so_far]:
                smallest_so_far = choice
        temp = alist[phase]
        alist[phase] = alist[smallest_so_far]
        alist[smallest_so_far] = temp
    return alist
```

Case study: Packing luggage

You would like to bring various supplies to your family overseas, but you are constrained by the weight limit for your luggage. What should you pack and what should you leave behind?

Aside

Later in the course we will consider other variants of this problem.

Fractional knapsack

Fractional knapsack

Input: A set of n types of objects, where object i has integer weight w_i and integer value v_i, and an integer weight bound W

Output: A list of fractions $0 \leq x_i \leq 1$ such that $\sum_{i=1}^{n} x_i \cdot w_i \leq W$ and $\sum_{i=1}^{n} x_i \cdot v_i$ is maximized

Example: $n = 3$, $W = 70$

i	w_i	v_i
1	50	100
2	30	30
3	40	4

Greedy algorithm for knapsack

Recipe for a greedy algorithm:

1. Build up the solution step by step. *Add one item at a time.*
2. Choose an option that is best at the moment. *Based on an ordering of the items, add the next item in order.*
3. Augment the solution using the chosen option. *Add as much of the item as will fit.*
4. Define options for the next step. *Reduce the remaining weight by the amount of the item added.*
5. Repeat the process without ever undoing a decision. *Never remove an item from the solution so far.*

Choosing highest value first

Order the objects from highest value to lowest
Take as much as possible of the objects in order

Example: $n = 5$, $W = 100$

i	w_i	v_i
1	10	20
2	20	30
3	30	66
4	40	40
5	50	60

Choosing lowest weight first

Order the objects from lowest weight to highest
Take as much as possible of the objects in order

Example: $n = 5$, $W = 100$

i	w_i	v_i
1	10	20
2	20	30
3	30	66
4	40	40
5	50	60

Greedy solutions compared

Choosing highest value first gives a total of 146:

Choosing lowest weight first gives a total of 156:

The best solution is a total of 164:

Choosing best ratio of value to weight

Idea: Get as much value as possible for each unit of weight.

Order the objects by nonincreasing ratio of value to weight
Take as much as possible of the objects in order

Running time:

Cost of sorting: $O(n \log n)$
Cost of trying objects: $O(n)$

Case study: Mapping out routes

As a frequent traveller, you wish to save money by determining the cheapest way of getting to each of the destinations you visit.

Single source cheapest paths

Input: A graph G with non-negative edge weights and a source vertex $s \in V(G)$

Output: Least-cost paths from s to each vertex in G, where the cost is the sum of the weights of edges in the path

Dijkstra's algorithm

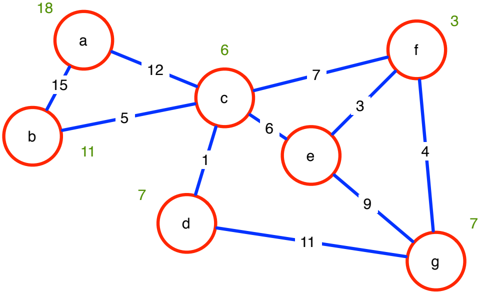
Maintain:

- K, the set of vertices such that cheapest paths from s are known
- U, the set of vertices not in K
- $\text{dist}(v)$ for all $v \in U$, the cheapest cost of a *special path* from s to v using only vertices in K as internal nodes on the path

Algorithm idea:

- Repeatedly (until U is empty), choose a v in U with the cheapest special path.
- Add v to K
- Update values of dist for all remaining vertices in U

Example of Dijkstra



Case study: Designing a network

You wish to set up a communications network that allows you to reach each entity from each other entity. There are costs associated with links between pairs of entities. How can you find a cheapest way of forming a network?

Minimum spanning tree

Definitions

A *spanning tree* is a tree connecting all vertices in a connected graph, formed of a subset of the edges in the graph. A *minimum spanning tree* is a spanning tree such that the sum of the weights of the edges is minimized.

Minimum spanning tree

Input: A connected graph G with non-negative edge weights

Output: A subset A of $E(G)$ that forms a tree from all of $V(G)$ and the sum of the weights of edges in A is as small as possible

Greedy approach to minimum spanning tree

Recipe for a greedy algorithm:

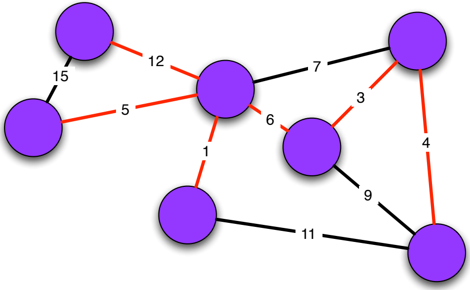
1. Build up the solution step by step. *Add one edge at a time.*
2. Choose an option that is best at the moment. *Choose an edge that is cheapest according to some criterion.*
3. Augment the solution using the chosen option. *Add the chosen edge to the solution so far.*
4. Define options for the next step. *Restrict edges depending on how the chosen edge was chosen.*
5. Repeat the process without ever undoing a decision. *Never remove an edge from the solution so far.*

1 $A \leftarrow \emptyset$
2 **while** *A does not form a spanning tree*
3   *find an edge e of least cost with certain properties*
4 $A \leftarrow A \cup \{e\}$

We will consider various options for line 3.

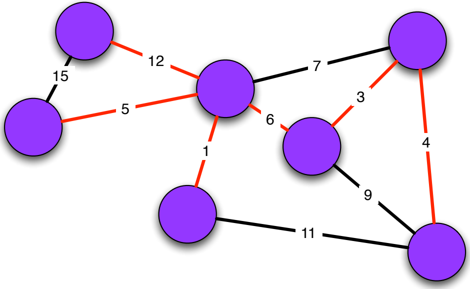
Kruskal's Algorithm

- Start with the empty set of edges
- Add the lowest weight edge that does not form a cycle.



Prim's Algorithm

- Start with a single vertex.
- Add the lowest weight edge that joins a vertex outside of the tree formed by A to the tree formed by A.



Analysis of minimum spanning tree algorithms

Sophisticated ways of manipulating data are used keep the running time low.

Details are outside the scope of this course.

Kruskal's algorithm: $O(m \log m)$ time

Prim's algorithm: $O(m \log n)$ time

Case study: Colouring a map

You wish to colour your historical map of the world in such a way that no two adjacent countries receive the same colour.

A map can be expressed as a graph.

To colour a map, each vertex represents a country, and an edge indicates that the countries corresponding to its endpoints are adjacent. The endpoints of an edge must be assigned different colours.

We can also look at colouring of graphs that do not represent maps.

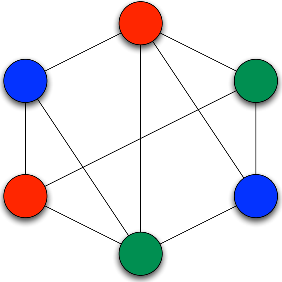
Colouring

A graph is ***k*-colourable** if each vertex can be assigned one of k colours such that each edge has endpoints of two different colours.

k-COLOURING

Input: A graph G and an integer k

Output: Yes or no, answering "Is G is k-colourable?"



Greedy 3-colouring

3-COLOURING

Input: A graph G

Output: Yes or no, answering "Is G is 3-colourable?"

Notice that the number three is not part of the input.

We may consider greedy approaches to this problem in assignments and/or exams.

Comparing paradigms

Aspect	Exhaustive search	Greedy
Feasible solutions	All	Some
Applicability	Wide	Narrow
Speed	Slow	Fast

Properties of greedy algorithms:

- Each step of the algorithm eliminates some possible solutions from consideration, as no decision is ever "undone"
- Running time analysis is often simple
- Proving correctness can be difficult
- Proving an algorithm is not correct can be achieved with a single counterexample

Module summary

Topics covered:

- Paradigm: Greedy algorithms
- Greedy TSP
- Analysis of greedy algorithms
- Proof by counterexample
- Case study: Scheduling activities
- Proof by contradiction
- Proof of optimal substructure
- Making change
- Greedy sorting
- Case study: Packing luggage
- Case study: Mapping out routes
- Case study: Designing a network
- Case study: Colouring a map
- Comparing paradigms