DSC 190

Data Structures & Algorithms

Today's Lecture

Algorithms

- We've been studying data structures.
- We'll now move towards algorithm design.
- Data scientists do design algorithms.
- But perhaps more important to understand solutions to common problems and which problems are difficult.

Today

- We'll introduce the idea of an **optimization problem**.
- Talk about one easy strategy that sometimes works.

DSC 190

DATA STRUCTURES & ALGORITHMS

Optimization Problems and Design Strategies

Optimization Problems

- We often want to find the **best**.
  - Shortest path between two nodes.
  - Minimum spanning tree.
  - Schedule that maximizes tasks completed.
  - Line of best fit.

- These are **optimization problems**.

Example: Regression

- Given a set of n points in $\mathbb{R}^2$, find a straight line $y = mx + b$ which minimizes the Sum of Squared Errors.

- **Given**: set of n points $\{(x_i, y_i)\}$ in $\mathbb{R}^2$

- **Search Space**: all straight lines of form $y = mx + b$

- **Objective Function**: $\phi(m, b) = \sum_{i=1}^{n}(y_i - (mx_i + b))^2$

Continuous Optimization

- Here, the search space is continuous, often **infinite**.
- Methods for solving often use calculus.

Discrete Optimization

- Here, the search space is discrete, typically **finite**.
- Example: shortest path between two nodes.
- Methods for solving (usually) can't use calculus.
- We will focus on these problems.

Brute Force

- If search space is finite, can employ **brute force search**.
- Typically search space is too large to be feasible.

Design Strategies

- Focus on **design strategies** for discrete optimization.:
  - **Greedy Algorithms**
  - **Backtracking**
  - **Dynamic Programming**

DSC 190

DATA STRUCTURES & ALGORITHMS

The Greedy Approach by Example

Problem

- **Choose**: 4 numbers with largest sum.

95	83	80	77
62	65	55	75
85	91	70	74
88	72	59	79

Specification

- **Given**: A set X of n numbers and an integer k.
- **Search Space**: Subsets $S \subset X$ of size k.
- **Objective**: maximize sum of numbers in S,

$$\phi(S) = \sum_{s \in S} s$$

Brute Force

- Brute force: try every possible subset of size k.

- How many are there?

$$\binom{n}{k} = \Theta(n^k)$$

- Time complexity is $\Theta(k \cdot n^k)$

The Greedy Approach

95	83	80	77
62	65	55	75
85	91	70	74
88	72	59	79

The Greedy Approach

- At every step, make the best decision at that moment.
- Is this optimal? Not always, but it is here.

Proof

Let $x_1 \geq \cdots \geq x_k$ be the k largest numbers. Let $y_1 \geq \cdots \geq y_k$ be some other solution. Since $x_1, \ldots, x_k$ are the k largest:

$$x_1 \geq y_1, \quad x_2 \geq y_2, \quad \ldots, \quad x_k \geq y_k.$$

Therefore:

$$\sum_{i=1}^{k} x_i \geq \sum_{i=1}^{k} y_i$$

Since the other solution was arbitrary, this shows that the greedy solution is at least as good as anything else; therefore it is maximal.

Efficiency

- Algorithm: loop through once, find k largest numbers.
- Linear time, $\Theta(n)$.
- Much faster than $\Theta(k \cdot n^k)$!

A Variation

- Now you can only choose one number from each row.

95	83	80	77
62	65	55	75
85	91	70	74
88	72	59	79

Specification

- **Given**: An $n \times n$ matrix X of numbers and an integer k.
- **Search Space**: Subsets $S \subset X$ of size k where each element is from a different row of X.
- **Objective**: maximize sum of numbers in S.

$$\phi(S) = \sum_{s \in S} s$$

Optimality

- The greedy approach of choosing largest within each row is optimal.

Another Variation

- Now you can only choose one from each row/column.

95	83	80	77
62	65	55	75
85	91	70	74
88	72	59	79

Specification

- **Given**: An $n \times n$ matrix X of numbers and an integer k.
- **Search Space**: all subsets of entries of X of size k such that each element is in a different row/column of X.
- **Objective**: maximize sum of numbers in subset.

$$\phi(S) = \sum_{s \in S} s$$

Greedy is not Optimal

- The optimal solution is: 80 + 75 + 91 + 88 = 334

95	83	80	77
62	65	55	75
85	91	70	74
88	72	59	79

Main Idea

For some problems, a greedy approach is guaranteed to find the optimal solution. For other problems, it is not.

Main Idea

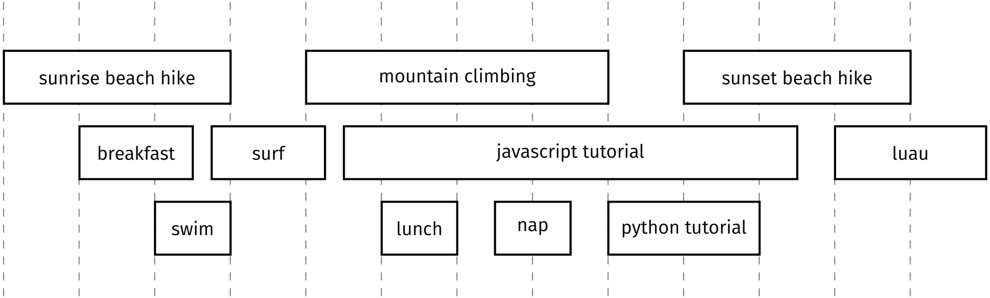
Coming up with a greedy algorithm is usually simple – proving that it finds the optimal may not be so easy.

DSC 190

DATA STRUCTURES & ALGORITHMS

Activity Selection Problem

Vacation Planning



Formalized

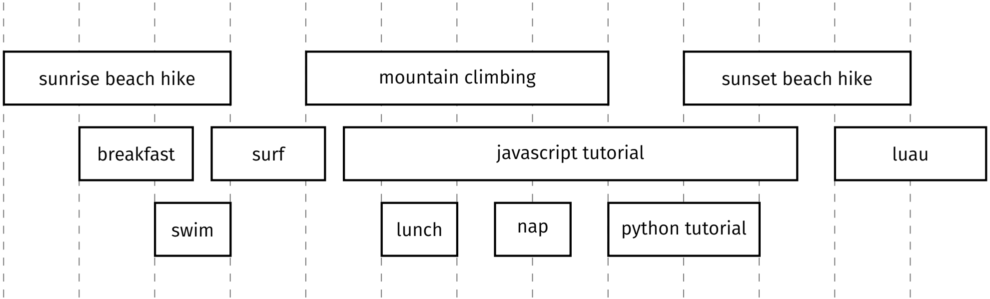
- This is called the **activity selection** problem.
- **Given**: a set of start/finish times (s_i, f_i) for n events
- **Search Space**: all schedules S with non-overlapping events
  - Format: S is a set of event indices $e_1, e_2, \ldots, e_k$
- **Objective**: maximize $|S|$ (number of events)

$$\phi(S) = |S|$$

Greedy Strategies

- There are several strategies we might call “greedy”.
- Approach #1: in order of duration, shortest events first.

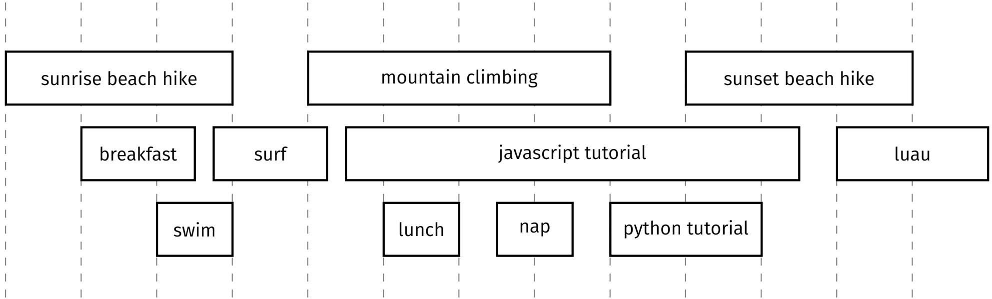
In Order of Duration



Greedy Strategies

- Approach #2: in order of start time.

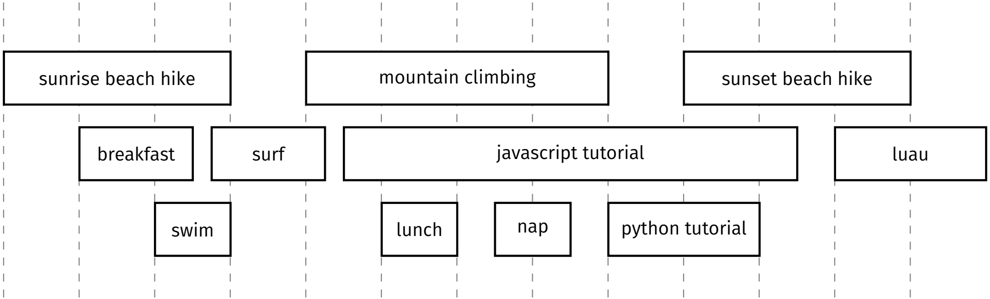
In Order of Start Time



Greedy Strategies

- Approach #3: in order of finish time.

In Order of Finish Time



In Order of Finish Time

- Choose event with earliest finish time as first event.
- Choose subsequent events in order of finish time.
  - provided that they are non-overlapping.
- This is **guaranteed** to find global optimum.
- But how do we know this?

DSC 190

DATA STRUCTURES & ALGORITHMS

Exchange Arguments

Convincing Yourself

- Designing a greedy algorithm is usually easy.
- It can be hard to convince yourself that it is optimal.
- Now, one proof technique: **exchange arguments**.

First: Proving **Non-Optimality**

- To show that a strategy is **non-optimal**, find a counterexample.

Proving Optimality

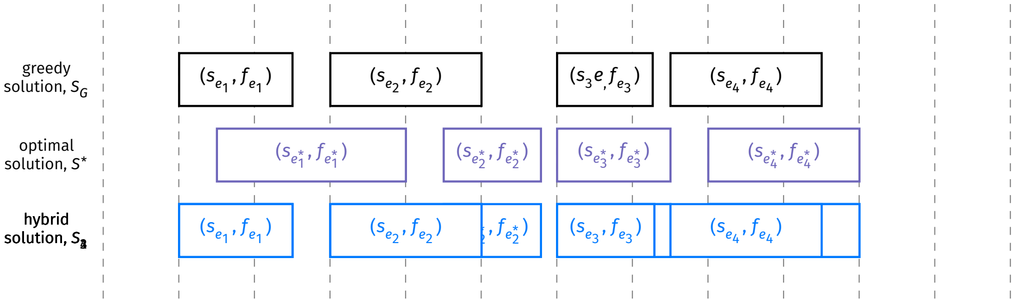
- There may be many optimal solutions – we want to show that the greedy solution S_G is always one of them.

Exchange Arguments

- Start with an arbitrary optimal solution, S^*.
- Make a **chain** of optimal solutions $S^*, S_1, S_2, \ldots, S_G$
- At every step from S_{k-1} to S_k:
  - construct solution S_k by **exchanging** part of S_{k-1} with S_G
  - argue that S_k is **valid**[1]
  - argue that S_k is **also optimal**
- Proves S_G is optimal, as $\phi(S^*) = \phi(S_1) = \phi(S_2) = \ldots = \phi(S_G)$

[1] It is part of the search space and meets all constraints.

Exchange Argument for Activities



Exchange Argument for Activities

Take an arbitrary optimal solution S^*. Suppose it is different from the greedy solution, S_G (as otherwise we're done).

If it's different, it has to be different *somewhere*. Let's look at the first event in S^* that is not in S; call this the ith event in S^*.

We'll exchange the ith event in S^* with the ith event in S_G, but we have to be a little careful: what if $|S^*| > |S_G|$, so that it's possible that S_G has no ith element? So there are two cases: $i \leq |S_G|$ and $i > |S_G|$.
First case: $i \leq |S_G|$. Then exchange the ith event in S^* with the ith event in S_G, creating a new solution S'.

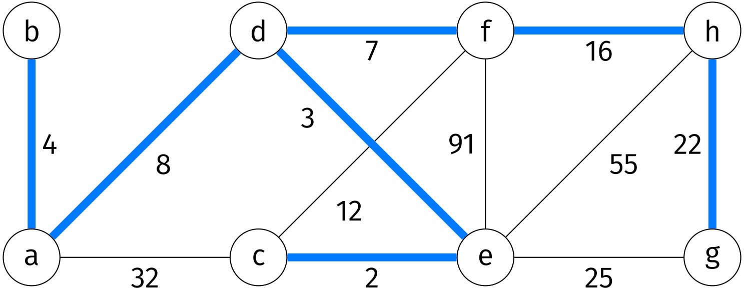
This is **valid**: the event from S_G cannot overlap with any of the events in S^*, since the previous $i - 1$ events in S^* are the same as in S_G (and they didn't overlap), and the finish time of the greedy event is $\leq$ the finish time of event it is replacing, so it cannot overlap with the remaining events.

It is also **optimal**, since $|S'| = |S^*|$.

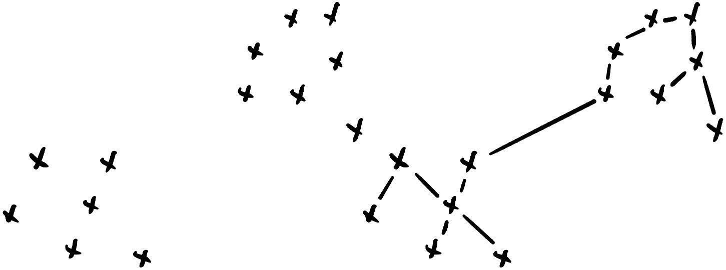
DSC 190

DATA STRUCTURES & ALGORITHMS

Minimum Spanning Trees


b
d
f
h
7
16
4
8
3
91
22
55
12
a
c
e
g
32
2
25

MSTs and Clustering



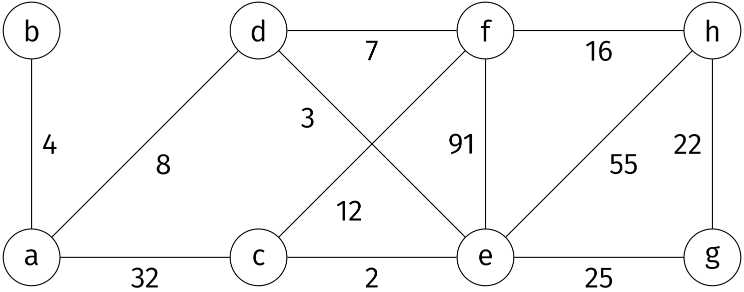
Minimum Spanning Trees

- **Given**: a weighted graph $G = (V, E, \omega)$, where $\omega : E \to \mathbb{R}$.
- **Search Space:** all **spanning trees** $T = (V, E')$, where $E' \subset E$.
- **Objective**: minimize total edge weight

$$\phi(T) = \sum_{e \in E'} \omega(e)$$

Kruskal's Algorithm

- **Kruskal's Algorithm** is a **greedy** algorithm for computing a MST.
- Idea: add edges one-by-one in order of weight.
    - But only if edge does not make a cycle!


b
d
f
h
7
16
4
8
3
91
55
22
12
a
c
e
g
32
2
25

Kruskal's Algorithm (Pseudocode)

```
def kruskals(graph, weight):
    mst = UndirectedGraph()
    edges = sorted(graph.edges, key=weight)

    for (u, v) in edges:
        if u and v are not connected in mst:
            mst.add_edge(u, v)

    return mst
```

Implementing Kruskal's Algorithm

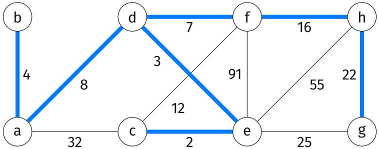
```
def kruskals(graph, weight):
    mst = UndirectedGraph()
    edges = sorted(graph.edges, key=weight)

    dsf = DisjointSetForest()
    for i in range(len(graph.nodes)):
        dsf.make_set()

    for (u, v) in edges:
        if dsf.find_set(u) != dsf.find_set(v):
            mst.add_edge(u, v)
            dsf.union(u, v)

    return mst
```

Optimality

- Kruskal's Algorithm find an optimal solution.
- We can prove this with an **exchange argument**.

Notes

- The greedy approach produces a valid spanning tree.
- Any two spanning trees have same number of edges.
- Removing an edge from a MST partitions nodes in two.



Exchange Idea

- Suppose $e^* = (u, v)$ is in T^*, but not in T.
- We'll find a node e on the path from u to v in T.
- Make a new tree, T', by taking T^*, removing e^*, replacing it with e.

Exchange Argument

Let T^* be any minimum spanning tree, and let T_G be a tree produced by Kruskal's algorithm. Suppose that T^* and T_G are different, and let $e^* = (u, v)$ be an edge in T^* that is not in T_G.

Consider the path from u to v in T_G. Adding e^* to T_G would create two different paths from (u, v), and thus a cycle. Let (A, B) be the cut produced if e^* were removed from T^*, and let e be an edge along the cycle that crosses the cut (A, B) (there must be at least one).

We will exchange e^* in T^* for the edge e.
First, this will create a **valid** spanning tree. Removing e^* in T^* breaks the tree into two connected components with disjoint node sets A and B. Since e crosses (A, B), adding it will re-connected the disconnected components, and thus form a spanning tree, T'.

Second, the new tree is **also optimal**. We claim that $\omega(e') \geq \omega(e)$. At the time e' was considered by Kruskal's, it was rejected because it would create a cycle. Meaning that edge e was already added, implying that $\omega(e) \leq \omega(e^*)$. As

DSC 190

DATA STRUCTURES & ALGORITHMS

Designing Greedy Algorithms

Designing Algorithms

- When do we know to use a greedy algorithm?
- It isn't always obvious.

A Pattern

- Our examples have a common pattern: sort by some attribute, then loop through.
  - Number grid: take numbers in descending order.
  - Activities: take activities in increasing order of finish time.
  - MST: take edges in increasing order of weight.

- This is a new justification for value of sorting.

- Suggestion: when tackling a problem, try sorting first.

Greedy Approximations

- A greedy algorithm can be useful, even if not guaranteed to produce optimal answer.
- Especially true if exact algorithms are **slow**.
- Example: k-means clustering (Lloyd's algorithm)

k-means Problem

- **Given**: n data points X in $\mathbb{R}^d$, parameter k.
- **Search Space**: all clusterings $C = \{X_1, \dots, X_k\}$ of X into k disjoint sets.
- **Objective function**: minimize

$$\phi(C) = \sum_{i=1}^{k} \sum_{x \in X_i} (x - \mathrm{mean}(X_i))^2$$

Greedy Algorithm

- Lloyd's algorithm (a.k.a., the "k-means algorithm") is a greedy algorithm for minimizing the k-means objective.
- Start with k centroids, $\mu_1, \dots, \mu_k$.
- At each step, let X_i be set of points closest to μ_i, update μ_i to be mean(X_i), repeat until convergence.
- Each step decreases value of objective function.

Optimality

- Lloyd's algorithm is **not** guaranteed to find optimum.
- Then again, no feasible algorithm is.
- Used in practice because it is fast and "good enough".