Greedy Algorithms

What is a Greedy Algorithm?

Kruskal's and Dijkstra's algorithms are greedy algorithms.

- A Greedy algorithm is very much like a Dynamic Programming algorithm, with one significant difference.
- A Dynamic Programming algorithm says "there's a bunch of choices, I don't know which one is correct, let's try them all and take the best!"
- A Greedy algorithm says "there's a bunch of choices, but I only need to try this specific one."

You apply a greedy criteria such as "always take the smallest-cost edge that doesn't create a cycle" or "choose the edge that minimizes distance-so-far + edge-cost".

Unweighted Interval Scheduling

In the unweighted interval scheduling problem, you are given n jobs.

- Each job j has a start time s_j, and a finish time f_j.
- Two jobs are compatible if there is no overlap in their time-spans.
- You want to choose the maximum number of mutually-compatible jobs.

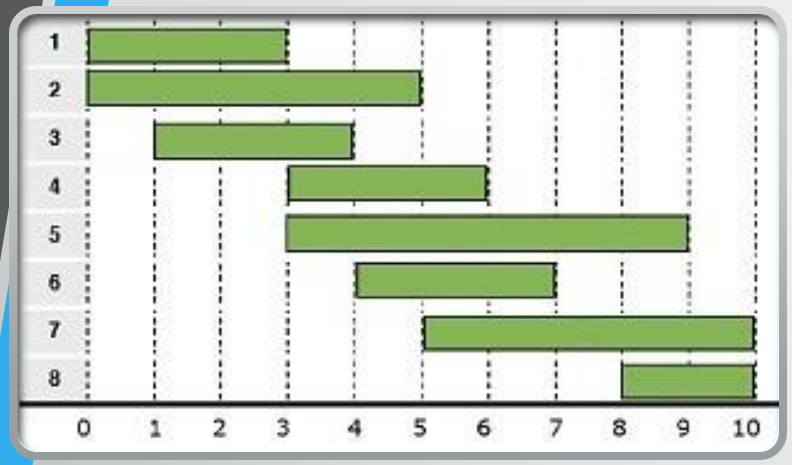
How is this different than the problem we already solved?

- It's unweighted: every job costs the same.

So what? We can just use the old dynamic programming algorithm.

- We might be able to make a more efficient algorithm!

Interval Scheduling: an Example



What is the best solution?

- 1, 4, 8
- 3, 6, 8

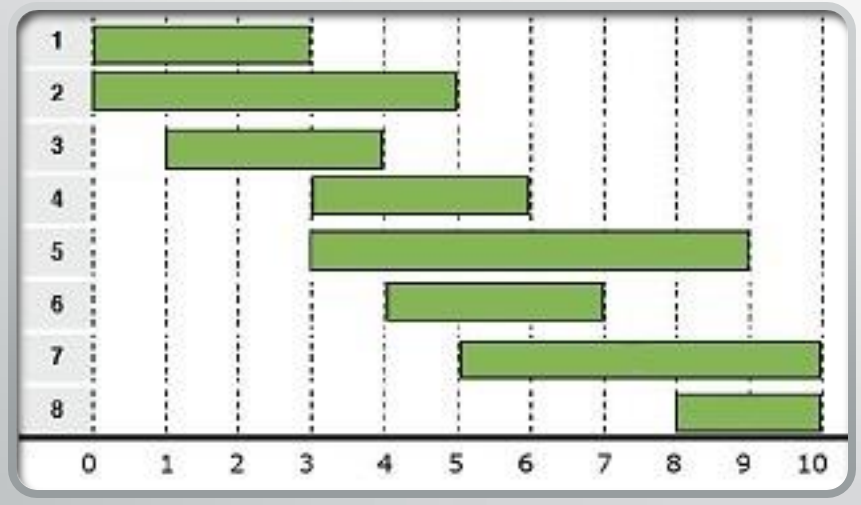
Are you convinced this is the best solution?

- If so, that's a sign that there's a simple underlying greedy algorithm!

Our dynamic programming algorithm asked "which job should I do next?"

Our greedy algorithm will ask the same thing, but it will figure out the answer without trying all possible solutions. It will apply a greedy criteria to arrive at the answer.

Interval Scheduling: Greedy Criteria



What are some possible greedy criteria for this problem?

- Earliest Start Time First
- Earliest Finish Time First
- Latest Start Time First
- Smallest Interval First
- Fewest Collisions First

Our provided example disproves Earliest Start Time First, as we might choose interval 2 first.

Disproving Smallest Interval First

Smallest Interval First will take the blue interval.

The correct solution is to take the black and red intervals.

Disproving Fewest Collisions First

Fewest Collisions First will take the green interval first.

The correct solution takes the top-row intervals

Two Algorithms Remaining…

Latest Start Time and Earliest Finish Time are mirrors of each other.

- Take any instance I, and reverse it to produce I', where all the start times become finish times and vice versa.
- Solving I with Latest Start Time is identical to solving I' with Earliest Finish Time, and vice versa.
- Therefore, a counter-example for one provides a counter-example for the other.

These algorithms are either both right, or both wrong.

Earliest Finish Time

While this algorithm is the last one standing, that doesn't mean it actually works.

- We may not have been clever enough, and entirely missed the correct algorithm.
- This problem may not have a greedy solution at all.

Greedy algorithms are easy to design, easy to write, and very efficient.

- The downside is that they are very unclear. **Why** does a given greedy algorithm actually work?

We will need to prove the correctness of this algorithm, using Induction.

- Just as Dynamic Programming and Greedy Algorithms broke the problem into bite-size decisions, so too will our proof.
- We will prove that our first choice was correct, then our second choice, then our third choice, etc. Inductively.

Exchange Arguments

What does it mean for our first choice to be correct?

- There is an optimal solution which includes our first choice.
- There may be many optimal solutions, we just care that one of them includes this choice. If none do, our choice was clearly wrong.

So, if our first two choices are correct, there must be an optimal solution that includes both of our first two choices.

- Our inductive hypothesis will (always) be "there is an optimal solution, called OPT, that includes our first k choices."
- Our inductive step will be to find an optimal solution, called OPT', that includes our first k+1 choices.
- Our base case will always be simple: "there is an optimal solution that includes our first 0 choices" (this is always vacuously true).

Earliest Finish Time: the Proof

Base Case: There is an optimal solution that includes EFT's first o choices (vacuously true).

Inductive Hypothesis: There is an optimal solution OPT that includes EFT's first k choices.

Inductive Step: Assume that OPT does not include EFT's $(k+1)^{st}$ choice (otherwise we're done).

- Sort OPT's intervals by finish time: EFT's first k choices must be the first k intervals in this sorting.
  - If there is an interval in OPT with earlier finish time, EFT would have chosen that instead.
- Let interval j be the $(k+1)^{st}$ interval in this sorting, and let interval i be EFT's $(k+1)^{st}$ choice.
- $f_i \leq f_j$, otherwise EFT would have chosen j instead of i.
- Transform OPT into OPT' be replacing interval j with i.
- Since i starts after the k^{th} interval in the sorting finishes, and finishes before j did, this must be a valid solution.

Sequential Exchange Arguments

1. There is an optimal solution which includes our first 0 choices (vacuous base case)
2. Assume there is an optimal solution OPT which includes our first k choices
3. Swap something in OPT with out $(k+1)^{st}$ choice to produce OPT'.
   - Figuring out what to swap is a large part of the challenge.
   - Simply saying "OPT's $(k+1)^{st}$ choice" is meaningless. OPT is a solution. It includes some choices, and not others. It doesn't specify the order to choose those choices, because the order is irrelevant to whether or not the solution is optimal.
   - You **can** impose an order on OPT's choices (such as sorting them by finish time), and use that to bring meaning to the phrase "OPT's $(k+1)^{st}$ choice". Just be cautious if you say this!
4. Prove that OPT' is still valid, that is it doesn't break any rules of the problem
5. Prove that OPT' is still optimal, that is it still optimizes the solution.

Kruskal's Algorithm

The **cycle property** states that the largest-cost edge in a cycle is not in the MST.

You may have seen a proof of Kruskal's Algorithm, which proved the cycle property when all edges have distinct weights.

- Assume edge f is in the MST T, and is the largest-cost edge in cycle C.
- Take T-f, which splits the tree T into two pieces. Part of C is in one side, and the rest of C is in the other
- Therefore, there is another edge e in C which reconnects the tree, and it costs less than f
- Since Kruskal's adds all edges except the largest-cost edges of cycles, Kruskal's is proven.

Kruskal's: an Exchange Argument

We will extend the prior proof to also work when the edge costs are not distinct. At its core, the proof is an **exchange argument**.

Base Case: There is an optimal solution that includes the first o edges from Kruskal's Algorithm (KA)

Inductive Hypothesis: Assume there is an optimal solution OPT that includes the first k edges from KA.

Kruskal's: the Proof

Inductive Step: KA includes edge i next: assume that OPT does not include this edge (otherwise we're done).

- OPT+i creates a cycle C. Since KA included edge i, there must be another edge in the cycle, j, that KA didn't include.
- It must be that $c_j \geq c_i$, because otherwise we would have considered, and included, j instead of i.
- OPT' = OPT+i-j costs no more than OPT, so it is optimal.
- OPT' has n-1 edges, because we added one and removed one.
- OPT' is connected: any pair of nodes that used edge j can use the rest of cycle C as a detour instead.
- Therefore OPT' is still a tree, and thus is valid.

Prim's Algorithm

The **cut property** states that for any partition ⟨P, V-P⟩ of the vertices, the cheapest edge spanning the partition is in the MST

You may have seen a proof of Prim's Algorithm, which proved the cut property when all edge costs are distinct.

- Assume edge e, which is the cheapest edge spanning ⟨P, V-P⟩, is not in T.
- T+e creates a cycle C. Part of the cycle is in P, the rest is in V-P.
- Therefore, there is another edge f that spans P and V-P, and e costs less.
- T+e-f is therefore a better MST.
- Since Prim's only adds the min-cost edge spanning the discovered and undiscovered nodes, Prim's Algorithm is proven.

Prim's: an Exchange Argument

We will extend the prior proof to also work when the edge costs are not distinct. At its core, the proof is an **exchange argument**.

Base Case: There is an optimal solution that includes the first o edges from Prim's Algorithm (PA)

Inductive Hypothesis: Assume there is an optimal solution OPT that includes the first k edges from PA.

Prim's: the Proof

Inductive Step: PA includes edge i next: assume that OPT does not include this edge (otherwise we're done).

- OPT+i creates a cycle C. Since PA included edge i, all other edges spanning the discovered and undiscovered nodes must cost at least as much.
- Since part of C is in the discovered nodes, and the rest is in the undiscovered nodes, there must be an edge j in C that spans the partition.
- OPT' = OPT+i-j costs no more than OPT, so it is optimal.
- OPT' has n-1 edges, because we added one and removed one.
- OPT' is connected: any pair of nodes that used edge j can use the rest of cycle C as a detour instead.
- Therefore OPT' is still a tree, and thus is valid.

Scheduling to Minimize Lateness

You have n tasks.

- Each task j has a duration t_j and a deadline d_j.

You must choose an order to execute the tasks.

- Once a task is started, you do not interrupt it until it is finished.
- As soon as the previous task finishes, the next task is started.

If a task finishes after its deadline, it is late.

Lateness

If you start task i at s_i, then it will finish at $f_i = s_i + t_i$.

- The lateness is $L_i = \max(0, f_i - d_i)$
- The lateness of the latest task is $\max_i L_i$. We want to minimize this value.

We are **not** minimizing the sum of the latenesses.

Lateness: an Example

	Duration	Deadline
Job 1	1	2
Job 2	2	4
Job 3	3	6

What order should we execute the tasks?

- 1, 2, 3
- This is the only solution that allows all tasks to have lateness 0.

Lateness: Greedy Criteria

What would be some possible greedy criteria for this problem?

- Shortest duration first
- Earliest deadline first
- Smallest slack (d_i-t_i) first

Disproving Shortest Duration

	Duration	Deadline
Job 1	1	3
Job 2	2	2

If you execute task 2 first, you achieve 0 lateness for both tasks.

If you execute task 1 first (shortest duration first), you get a lateness of 1 for task 2.

Disproving Smallest Slack

	Duration	Deadline
Job 1	1	2
Job 2	3	3

If you execute task 1 first, you achieve only 1 lateness for task 2.

If you execute task 2 first (smallest slack first), you get a lateness of 2 for task 1.

Inversions

For a given schedule S, an **inversion** is a pair of jobs i and j, where the optimal solution schedules i before j, and S schedules j before i.

- If OPT has no inversions with your schedule S, then the two schedules are the same, which means S is optimal.

A **consecutive inversion** is a pair of jobs i and j, where S schedules j before i, and OPT schedules i immediately preceding j.

We will attempt to prove there is an optimal solution that has no inversions with our algorithm Earliest Deadline First (EDF).

We will do induction, removing one consecutive inversion at a time.

Lateness: an Exchange Argument

This is still an exchange argument: we are still swapping decisions of OPT for decisions of EDF, and showing that this maintains optimality.

Base Case: There is an optimal solution with $\leq C(n, 2)$ inversions (the max possible number of inversions). Vacuously true.

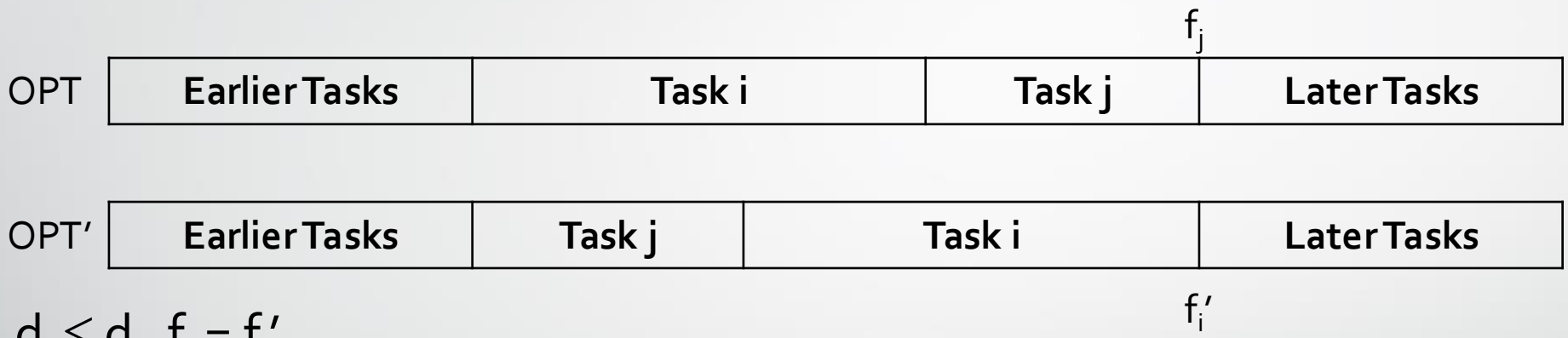
Inductive Hypothesis: Assume there is an optimal solution OPT, with $\leq k$ inversions.

Lateness: the Proof

Inductive Step: There is a consecutive inversion where OPT schedules i immediately before j, but EDF schedules j first.

- Transform OPT into OPT' by swapping i and j's order.
- Nothing before i is affected by this swap.
- Nothing after j is affected: they all still need to wait for both i and j to finish.
- j is only improved, as it moves earlier in the schedule.
- The only task that might (and probably will) get worse is task i.

Lateness: the Exchange



$d_j \leq d_i$, $f_j = f_i'$

$L_i' \leq L_j \leq OPT$

- Since no other task gets worse in OPT', it must still be optimal.
- We haven't broken any rules, so it is also valid.

Non-Consecutive Inversions

What if there is an inversion, but no consecutive inversions?

- That's actually impossible!

Assume that $\langle i, j \rangle$ is the smallest inversion, and task k is scheduled between those tasks by OPT.

- If we schedule k before i, then $\langle i, k \rangle$ is a smaller inversion
- If we schedule k after j, then $\langle k, j \rangle$ is a smaller inversion
- Don't forget we schedule j before i, so at least one of those two things must happen!

Inversion Exchange Arguments

You typically use these for **ordering** problems.

1. Vacuous Base Case: there is an optimal solution with no more than C(n, 2) inversions (which is the max possible number of inversions)
2. Inductive Hypothesis: assume there is an optimal solution OPT with no more than k inversions.
3. Inductive Step: remove a consecutive inversion from OPT to produce OPT'.
4. Prove that OPT' is still optimal.
5. Prove that OPT' is still valid.

Optimal Caching

A cache can store n items. There is a sequence of m requests d_1, …, d_m, known in advance.

- If an item is requested which is not in the cache, it must be brought into the cache, resulting in a **cache miss**.
- The goal is to minimize the number of cache misses.

Requests	a	b	c	b	c	a	a	b
Cache1	a	a	c	c	c	a	a	a
Cache2	b	b	b	b	b	b	b	b

Greedy Criteria

When a cache miss occurs, which item should you remove from the cache?

- Least Recently Used
- Least Frequently Used in Future
- Furthest in Future

The first two algorithms are disproved by our original example!

How is this problem different than the **real** caching problem?

- We know the requests in advance. The real caching problem can't use the 2nd or 3rd algorithm.

The Proof

Base Case: There is an optimal solution with the same cache contents as US through the first 0 requests.

Inductive Hypothesis: There is an optimal solution OPT with the same cache contents as US through the first k requests.

Inductive Step: Assume OPT has different cache contents than US at the $(k+1)^{st}$ request (otherwise we're done).

What must happen at request k+1?

- A cache miss, otherwise our two solutions can't diverge at this point.

The Exchange, Part 1

US	d_k	d_{k+1}	…
Cache1	c_f	d_{k+1}	
Cache2	c_o	c_o	
Cache3	c_e	c_e	
…			

OPT	d_k	d_{k+1}	…
Cache1	c_f	c_f	
Cache2	c_o	d_{k+1}	
Cache3	c_e	c_e	
…			

At request k+1, US kicks out cf, and OPT kicks out co. Otherwise, their cache contents are identical.

For the first part of the exchange, change OPT into OPT' by kicking out c_f instead of c_o.

Events

OPT' will copy OPT until an **event** occurs, wherein their different cache contents force these two solutions to behave differently.

- There is a cache request on the item OPT' has that OPT doesn't (c_o).
- There is a cache request on the item OPT has that OPT' doesn't (c_f).
- OPT kicks out the item that OPT' doesn't have (c_f).

Of these three possible events, one of them **cannot** be the first event. Which one?

- We cannot have a cache request on c_f before c_o. Otherwise US would have kicked out a different cache item. c_f was supposed to be the furthest-in-future.

Case 1

Suppose the first event, at request j, is when OPT removes the item OPT′ doesn't have (c_f).

- There was a cache miss at this time: OPT′ can remove the item OPT doesn't have (c_o).

OPT′	d_k	d_{k+1}	…	d_j	d_{j+1}	…
Cache1	c_f	d_{k+1}		$X=d_{k+1}$	X	
Cache2	c_o	c_o		$Y=c_o$	d_{j+1}	
Cache3	c_e	c_e		$Z=c_e$	Z	
…						

OPT	d_k	d_{k+1}	…	d_j	d_{j+1}	…
Cache1	c_f	c_f		$W=c_f$	d_{j+1}	
Cache2	c_o	d_{k+1}		$X=d_{k+1}$	X	
Cache3	c_e	c_e		$Z=c_e$	Z	
…						

OPT and OPT′ now have the same cache contents, and accomplished this with the same number of cache misses. OPT′ is optimal!

Case 2

Suppose the first event, at request j, is when there is a request on c_o.

- OPT will kick out something both OPT and OPT' have (c_e).

OPT'	d_k	d_{k+1}	…	d_j	d_{j+1}	…
Cache1	c_f	d_{k+1}		$X=d_{k+1}$	X	
Cache2	c_o	c_o		$Y=c_o$	Y	
Cache3	c_e	c_e		$Z=c_e$	Z	
…						

OPT	d_k	d_{k+1}	…	d_j	d_{j+1}	…
Cache1	c_f	c_f		$W=c_f$	W	
Cache2	c_o	d_{k+1}		$X=d_{k+1}$	X	
Cache3	c_e	c_e		$Z=c_e$	Y	
…						

- We can't kick out X, because we don't have a cache miss. However, we are now one cache miss ahead, and there is still only one cache item we disagree on.

Case 3

The first event was case 2, so we are now one cache miss ahead, and **now** there is a request on the item OPT has that we don't (c_f).

OPT'	d_k	d_{k+1}	…	d_j	d_{j+1}	…	d_i	d_{i+1}
Cache1	c_f	d_{k+1}		$X=d_{k+1}$	X		$A=d_{k+1}$	A
Cache2	c_o	c_o		$Y=c_o$	Y		$B=c_o$	D
Cache3	c_e	c_e		$Z=c_e$	Z		$C=c_e$	C
…								

OPT	d_k	d_{k+1}	…	d_j	d_{j+1}	…	d_i	d_{i+1}
Cache1	c_f	c_f		$W=c_f$	W		$D=c_f$	D
Cache2	c_o	d_{k+1}		$X=d_{k+1}$	X		$A=d_{k+1}$	A
Cache3	c_e	c_e		$Z=c_e$	Y		$C=c_e$	C
…								

- **Now** we can kick out the item OPT' has that OPT doesn't (c_o), leaving OPT and OPT' with the same cache contents and the same number of cache misses. OPT' is optimal!

The Big Picture

We will transform OPT into OPT', where OPT makes the same decisions US does through the k^{th} request, and OPT' makes the same requests US does through the $(k+1)^{st}$ request.

- We start by changing OPT's decision at the $(k+1)^{st}$ request to do what US does.
- Now OPT and OPT' disagree on a single cache item: OPT has W, OPT' has Y.
- If OPT then kicks out W, OPT' kicks out Y to resolve all differences.
- If there is a request on Y, OPT' is now one cache miss ahead, and they still only disagree on a single item: OPT has W, and OPT' has Z.
- Now there could be a request on W, at which point OPT' kicks out Z to resolve all differences.

Extra Practice

Chapter 4

Exercises 3, 5, 6, 7, 9, 13, 15, 24

Prove the correctness of your algorithms, using exchange arguments.