Reductions, NP-Completeness, and Dealing with Hard Problems

Doing Better

We often ask “can we do better?” – or “how do we do better?”

But – is it *possible* to do better?

A key strategy involves reductions.

Reductions

Three ingredients:

- turn an instance of your problem into an instance of the other problem
- solve the other problem
- turn the solution for the other problem into a solution for your problem

Specifying a reduction means specifying the first and third steps.

(“The other problem” is something with a known algorithm or a known running time.)

Reductions

Two main applications –

- finding an algorithm for a new problem
- proving hardness (in the sense of the time required to solve the problem)
  - so you don't waste time trying to find an efficient algorithm when there isn't one
  - understanding what makes a problem time consuming to solve (i.e. hard) can help with trying to find a better way to address it

Polynomial-Time Reductions

- turn an instance of your problem into an instance of the other problem
- solve the other problem
- turn a solution to the other problem into a solution to your problem

the time for the first and third steps is the time for the reduction

The reduction should be as efficient as possible.

- exponential time isn't efficient
- for an algorithm, an exponential-time reduction to a polynomial-time task doesn't help
- for hardness reductions, concluding that the problem itself is hard means that the reduction can't be the source of the exponential time

Polynomial-Time Reductions

The *n* queens problem could be reduced to a graph problem as follows:

Construct a graph with a vertex for each possible partial solution - a empty board, each of the possible placements for a queen in the first column, each of the possible pairs of placements for queens in the first and second columns, etc. Connect two partial solutions with a directed edge if adding a queen to the first partial solution yields the other. Finding a valid placement of queens then becomes a question of reachability - which, if any, vertices corresponding to states with *n* queens are reachable from the vertex corresponding to the empty board.

Is this a polynomial-time reduction?

Is the reduction in the previous problem correct? That is, will the proposed solution in terms of reachability give a correct answer to the *n* queens problem?

The *n* queens problem could be reduced to a graph problem as follows:

Construct a graph with a vertex for each possible position of a queen on the board (n^2 vertices, each with one queen), plus another vertex corresponding to an empty board. Connect two vertices with a directed edge if the first vertex has a queen in column *i* and the second vertex has a queen in column *i*+1 and the queens do not attack each other. Also connect the empty-board vertex to each of the vertices with a queen in column 1. Finding a valid placement of queens is a question of reachability - which, if any, vertices corresponding to queens in column *n* are reachable from the vertex corresponding to the empty board.

Is this a polynomial-time reduction?

Is the reduction in the previous problem correct? That is, will the proposed solution in terms of reachability give a correct answer to the *n* queens problem?

Reductions for Hardness

If problem A has a polynomial-time reduction to problem B, which of the following can you conclude?

- a polynomial-time algorithm for A means there's a polynomial time algorithm for B
- a polynomial-time algorithm for B means there's a polynomial time algorithm for A
- no polynomial-time algorithm for A means there's no polynomial time algorithm for B
- no polynomial-time algorithm for B means there's no polynomial time algorithm for A
- none of these

how to solve A?
- reduce to B
- some other algorithm

A reduced to B →
- instance of A is turned into instance of B
- B is solved
- instance of B's solution is turned into instance of A's solution

Graph Reductions

Modeling a problem as a graph problem often takes the form of a reduction.

- (with a reduction, the "solve the other problem" step is a known algorithm so the goal is to build a graph to use with a known algorithm instead of building a graph and possibly also coming up with a graph algorithm to find a solution in the graph)

Graph Reductions

A common pattern for graph reductions –

- vertices represent the things in the solution
- directed edges represent ordering constraints
  - idea is that a (weighted) path in the graph corresponds to a solution that respects the constraints

Another common pattern –

- vertices represent the things in the solution
- undirected edges represent the constraints
  - an edge between two vertices means the vertices can't coexist in the solution, no edge means they don't conflict
    - e.g. coloring, independent set
  - an edge between two vertices means the vertices are redundant in the solution, no edge means both may be needed
    - e.g. vertex cover

Longest Increasing Subsequence

Given a sequence S of numbers, find the longest subsequence containing increasing numbers. The numbers in the subsequence must occur in that order in S, but need not be consecutive in S.

- observation – a subsequence is an ordered thing, sounds like a path → solution-is-a-path pattern

- vertices – things in the solution
  - the elements of S
- directed edges – ordering constraints between things
  - connect elements S[*i*] and S[*j*] if S[*j*] > S[*i*] and *j* > *i*
  - observation: the graph is a DAG (DAG = directed acyclic graph)
- legal solution
  - a path in the graph corresponds to an increasing subsequence
- optimal solution
  - the longest path in the graph (a DAG)

Longest Path in a DAG

We need an algorithm to find the longest path in a DAG.

A related problem –

Given a (weighted) DAG, find the longest path from vertex *s* to all other vertices.

But this isn't quite our problem...

- we don't have a start vertex
- we want the longest of the longest paths – we don't know the end vertex

Solution – adapt the graph.

- add vertices *s*, *t* to the longest increasing subsequence graph
  - connect *s* to every vertex
  - connect every vertex to *t*

the longest path from *s* to *t* will contain the longest path in the original graph

Longest Increasing Subsequence

The full solution, via reduction –

- build graph G
  - vertices – the elements of S plus *s*, *t*
  - directed edges
    - connect vertices for S[*i*] and S[*j*] if S[*j*] > S[*i*] and *j* > *i*
    - connect *s* to every vertex
    - connect every vertex to *t*
- find longest path *p* from *s* to *t* in G
  - G is a DAG
    - no incoming edges for *s* or outgoing edges for *t*
    - no cycles involving the other vertices because a cycle would require a smaller value to follow a larger one
- drop *s* and *t* from *p*, then take the corresponding elements from the remaining vertices in *p* to get the longest increasing subsequence

Longest Increasing Subsequence

Running time?

- create graph
  - O(n) vertices
  - $O(n^2)$ edges
    → $O(n^2)$ total (assuming O(1) to add to graph)
- find the longest path from *s* to *t*
  - topological sort from *s* followed by visiting the vertices in reverse order and setting $dist[v] = \max \{ dist[u]+w_{vu} \mid (u,v) \in E \}$
    → O(|V|+|E|) = O(n+m)
- get the elements in the sequence from the longest path
  → O(n)

→ $O(n^2)$
  - (same as the dynamic programming solution)

Bigger-Is-Smarter Elephants

You have n elephants, each with weight, intelligence, and value. Find the highest-value sequence of elephants such that the elephants get smarter as they get bigger.

- legal solution
  - subset of elephants $e_1, e_2, \ldots, e_m$ such that $w_i < w_j$ and $s_i < s_j$ for $i < j$
- optimization goal
  - maximize total value of elephants

Can we use a graph here too?
- “sequence” sounds like it could be a path…

Bigger-Is-Smarter Elephants

You have *n* elephants, each with weight, intelligence, and value. Find the highest-value sequence of elephants such that the elephants get smarter as they get bigger.

- vertices – things in the solution
  - elephants
- directed edges – ordering constraints between things
  - connect elephant i to elephant j if j is bigger and smarter than i
- other elements
  - value of elephants → vertex weights
- legal solution
  - a path in the graph corresponds to a legal ordering of elephants
- optimal solution
  - max-weight vertex-weighted path in a DAG

Max-Weight Vertex-Weighted Path in a DAG?!

This isn't quite like the longest path in a DAG –

- vertex weights instead of edge weights
- don't have known start and end vertices

Adapt the graph slightly –

- handle lack of start/end vertices as before
  - add source *s* connected to everything and sink *t* that all other vertices connect to
- convert vertex weights to edge weights by pushing the weight onto each outgoing edge (that this works should be justified – think about why, for a path in the graph, the sum of the edge weights assigned according to this scheme and the sum of the vertex weights in the original graph are the same)

Now the longest path in a DAG algorithm can be used as is.

Bigger-Is-Smarter Elephants

Running time?

- create graph
  - O(n) vertices
  - $O(n^2)$ edges

    → $O(n^2)$ total (assuming O(1) to add to graph)
- find longest path from *s* to *t*

  → O(|V|+|E|) = O(n+m)
- get the sequence of elephants from the longest path

  → O(n)

→ $O(n^2)$

Driving to Seattle

You want to drive from Boston to Seattle with the fewest number of stops for gas. You can drive at most 400 miles on a tank of gas.

- legal solution
  - set of gas stations with no more than 400 miles between them
- optimization goal
  - minimize the number of stops for gas

Does a graph reduction work here?

- solution is a sequence of stops → solution-is-a-path pattern

Driving to Seattle

- vertices – things in the solution
  - gas stations + Boston, Seattle
- directed edges – ordering constraints between pairs of things
  - between *i* and *j* if *i*, *j* are ≤ 400 miles apart and *i* comes before *j* on the road
- legal solution
  - a path in the graph corresponds to a particular sequence of stops between Boston and Seattle
- optimal solution
  - shortest path (in terms of number of vertices) from Boston to Seattle
    - observe that this is equivalent to the shortest path in terms of the number of edges (number of edges in a path = number of vertices in the path - 1) → unweighted shortest path

Driving to Seattle

Running time?

- create graph
  - O(n) vertices
  - $O(n^2)$ edges
    → $O(n^2)$ total (assuming O(1) to add to graph)
- find shortest unweighted path (breadth-first search)
    → O(|V|+|E|) = O(n+m)
- find the sequence of stops from the shortest path
    → O(n)

→ $O(n^2)$
  - (not as good as the greedy solution)

Movie Scheduling

There are two timeslots in which movies can be shown, and each person wants to see two movies. Is there a way to schedule the movies so that each movie is only shown once but everyone gets to see both of their choices?

- observation – the goal is an assignment of labels (timeslots) to things (movies) → vertex coloring

- vertices – things in the solution
  - movies
- undirected edges – constraints
  - can't schedule someone's two choices on the same day → vertices = movies, edges = people
  - edge (*u*,*v*) indicates that someone wants to see both *u* and *v*
- legal solution
  - a 2-coloring (there are two timeslots)

Movie Scheduling #2

Each person has two choices for movies they'd like to see. What's the fewest number of movies that the theater needs to book so that everyone gets to see at least one of their choices?

- observation – doesn't sound like a solution-is-a-path pattern…

- vertices – things in the solution
  - either movies or people
- undirected edges – constraints
  - don't need to book both of someone's choices → vertices = movies, edges = people
  - edge (*u*,*v*) indicates that someone wants to see both *u* and *v*
- legal solution
  - vertex cover – edge between two vertices means that those vertices are redundant (only need one to satisfy that person)
- optimal solution
  - min vertex cover

Other Reductions

Graph reductions are not the only possible reductions.

Subset Sum

Given a set of numbers, determine if there is a subset of the numbers which sum to some target value *t*.

This reduces to the decision-problem version of 0-1 knapsack.

- 0-1 knapsack (decision version): is there a subset of items with total weight at most W such that the total value is at least V?

Reduction:

- number *i* → item with value *i* and weight *i*
- capacity of pack W = *t*
- desired total value V = *t*

Solution:

- a subset with sum *t* exists if and only if the 0-1 knapsack answer is yes
  - since each item's weight is the same as its value, the only way to have a total weight ≤ t and a total value ≥ t is for both to equal *t*

Longest Common Subsequence

Given sequences A and B, find the longest subsequence common to both. The elements of the subsequence need not be consecutive in A or B, but must appear in the same order in both.

This reduces to edit distance –

- compute the edit distance between A and B with the cost of insertions and deletions = 1 and the cost of substitutions = ∞
- longest common subsequence length = (|A|+|B|-edit distance)/2

(Delete everything in A not in the common subsequence, and insert everything in B not in the common subsequence. The only elements that incur no cost are those common to both A and B, which are counted twice in |A|+|B|.)

Longest Increasing Subsequence

Given a sequence S of numbers, find the longest subsequence containing increasing numbers. The numbers in the subsequence must occur in that order in S, but need not be consecutive in S.

This reduces to longest common subsequence –

- sort S (call this sequence T)
- find longest common subsequence of S and T

(T ensures that the common subsequence must also be increasing. S ensures that the elements in the common subsequence must appear in the proper order.)

Reductions for Algorithms

- can be helpful for solving a new problem
  - provides another way of thinking about the problem which may reveal new insights
  - can provide a black box for solving the trickiest algorithmic part
- but may not be the most efficient way to solve the problem
  - e.g. driving to Seattle O(n) greedy algorithm if sorted, O(n log n) if not → shortest path in a graph $O(n^2)$