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?

True
False

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?

True
False

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?

True
False

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?

True
False

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 *(no pair of vertices in the set connected by an edge)*
  - an edge between two vertices means the vertices are redundant in the solution, no edge means both may be needed
    - e.g. vertex cover *(every edge connected to at least one vertex in set)*

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} | (u,v) ∈ 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)

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|.)

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 is an O(n) greedy algorithm if sorted, O(n log n) if not → shortest unweighted path in a graph $O(n^2)$

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

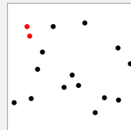
how to solve A?
- reduce to B
- some other algorithm

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

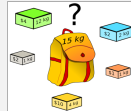
Complexity

Some problems seem to be more difficult to solve efficiently than others.

- the obvious brute force algorithm often has very different running time for different algorithms
  - e.g. closest pair of points – n^2
    - compute the distance for every pair



  - e.g. 0-1 knapsack – 2^n
    - try every subset



https://en.wikipedia.org/wiki/Closest_pair_of_points_problem
https://plus.maths.org/content/maths-minute-knapsack-problem

Complexity

Some problems seem to be more difficult to solve efficiently than others.

- small changes in a problem can make it much harder to solve
  - e.g. fractional knapsack vs 0-1 knapsack
  - e.g. linear programming vs integer linear programming
  - e.g. shortest path in a graph vs the longest
    - (note: general graph, not limited to DAG)
  - e.g. use every edge once (Euler circuit) vs use every vertex once (hamiltonian cycle)

Complexity

Some problems seem to be more difficult to solve efficiently than others.

- algorithmic techniques which work to speed up some problems don't work for others
  - e.g. greedy vs dynamic programming vs recursive backtracking

Complexity

Are there some problems which take fundamentally longer to solve than others, or have we just not been clever enough yet to find an efficient solution?