These notes describe the highlights of Chapter 8 of Dasgupta. The chapter contains many examples (mainly reductions) that are not included here. It is not compulsory to know all those examples. However, reading some of them may be helpful to understand the techniques introduced in this chapter.

First, read Section 0.3 on running time. Most importantly, we say that an algorithm runs in polynomial time if the running time is bounded by $p(|I|)$ where $p()$ is a polynomial and $|I|$ is the size of the input. By *efficiently* we shall mean in polynomial time.

Flavors of Optimization

Problems in combinatorial optimization typically contain the following three elements:

- A set of instances;
- For each instance, a set of candidate solutions.
- For each instance, a set of (valid) solutions. These are the candidate solutions with a certain property that we look for.

For example, in the Traveling Salesman Problem (TSP) each instance is a set of points with an integer distance $d(i, j)$ between every pair of points. Given an instance, a candidate solution is a tour going through each point exactly once, or in other words a permutation of the points. A solution could be a tour 'of minimal length' or 'of length at most 100'. (We shall always assume that given numbers are integer.)

Many optimization problems have this structure and for almost all natural problems it easy to check if a given candidate solution has the desired property. For example, we can easily check wether a given TSP tour has length at most 100 by adding up the distances on the tour. But what if the problem is to find *the* 2^n*-th shortest tour among all possible tours*? Even if we were given such a tour, how would we verify that it does indeed have this property? To avoid these strange problems (for which even verifying if a solution has the property is difficult) one makes the following restriction.

The class of search problems is defined as all problems for which *verifying* if a given candidate solution is indeed a valid solution *can be done in polynomial time.* The class of search problems is also known as NP.

Optimization versus Search By the definition above, the problem of finding the shortest TSP tour is *not a search problem.* Given an optimal solution, how should we check that there is indeed no shorter solution? There may not be an efficient way to do that. Our definition of search problems appears to have the flaw that finding an optimal solution is *not* in the class of search problems. However, this is just a technical formality since we can easily model any optimization problem as a search problem.

Search version of TSP: Given an instance of TSP and a number K we need to find a tour of length at most K if one exists.

Clearly, computing the length of a given tour is easy. Hence, this version of the TSP *does satisfy* the definition of a search problem. Moreover, if we have an algorithm that solves this search version, then we can also use that algorithm to find an optimal solution. This works as follows. (Remember that all distances are assumed integer.) First, we try to find a solution of length 1. If none exists then we try to find one of length 2, and so on. The moment that we do find a solution, it will be the shortest. Note that this can be done more efficiently (in polynomial time) by binary search. See Exercise 1.

Hence, every optimization problem can be modeled as a search problem. Moreover, any algorithm for the search problem can be used, with only a polynomial factor loss in the running time, to find an optimal solution.

Genuine search problems. Some search problems do not originate from optimization problems. For example, finding the prime factors of an integer number is not an optimization problem. (If the input is 70, the output should be $2, 5, 7$ since $2 \cdot 5 \cdot 7 = 70$. If the input is 31, the output should be 31, i.e., it is a prime.) But prime factorization is a search problem by our definition. Given a solution, e.g. $2, 5, 7$, we can easily check that the product is indeed 70. Another example is the Rudrata cycle problem. In instance is given by a graph (no distances!) and we search for a Rudrata cycle. (NB. Rudrata is *not* the search version of TSP. See the search version of TSP above.) Another example is graph isomorphism. See Figure 1. Given two

graphs, are they actually the same if we ignore the labelling of the vertices? Here, we search for a bijection between the vertex sets of the two graphs. (A

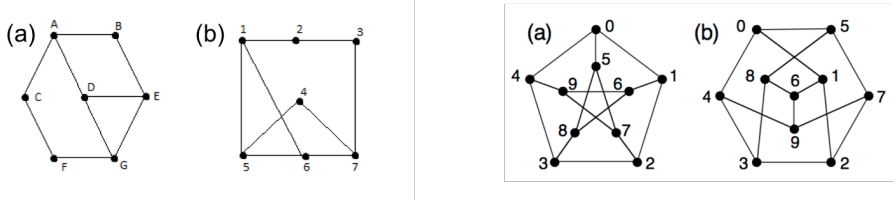


Figure 1: Graph Isomorphism: Are the two graphs on the left the same? How about the two on the right? (Answer: No, Yes)

correct bijection is shown in the rights pair). If two graphs are the same then a bijection exists and we can check equivalence easily by checking each of the edges. (Note that in the definition of search problems, we only require that verifying is easy for valid solutions, in this case a correct bijection. Verifying that two graphs are *not* the same may not be easy at all. In fact, it is a well-known open problem what the complexity of verifying the non-equivalence of two graphs is.)

Search versus decision. Every search problem has a natural decision variant. Here, we do not search for a valid solution (a proposed solution with a desired property) but only want to know if a solution exists. That means, we only require a correct yes/no answer: Is there a TSP tour of length at most 100? Are the two graphs the same? Does a number have more than one prime factor? If we have an algorithm for the search problem, then we obviously do have an algorithm for the decision version as well: If we find a desired solution then the answer is yes, and the answer is no otherwise. Maybe less obvious is that the other direction usually holds as well, i.e., an algorithm for the decision problem can be used to solve the search problem. (See Exercise 2.)

Beyond search problems Some natural problems do not fit in our definition of search problems by the fact that verification is hard or even impossible. Consider for example the *tiling problem* as illustrated in Figure 3. If an infinite covering is possible then this can be verified if the structure repeats itself. The left triple has a solution of 9 tiles. (Solution on the last page.) However, one can prove that there are instances that do allow

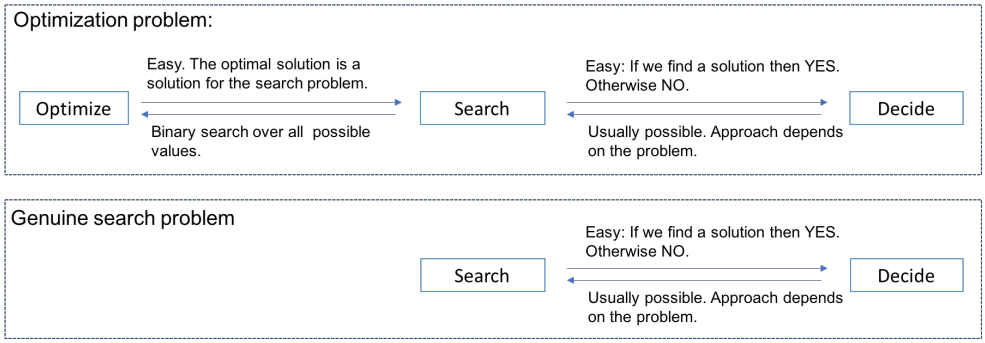


Figure 2: Three forms of optimization. Some search problems do not have an optimization variant.

a covering of the plane but for which any covering is non-repetitive. How would be we able to verify a solution that has no finite description? The tiling problem is undecidable: there just does not exists an algorithm for it, even if the algorithm was given unlimited computing power.

Another example is the *halting problem*: Given an algorithm and an input, can we decide wether the algorithm will terminate or wether it will run forever? No, we cannot. There just is no way to decide this, even if we had unlimited computing power.

Figure 3: Tiling: Can we cover the whole infinite plane with copies of the three tiles left? How about the three on the right? (Touching sides should match and rotation is not allowed.) (Answer: Yes, No)

P versus NP

The class of all search problems is denoted by NP. Hence, these are problems where we search for a solution and for which we can verify in polynomial time that a *given* proposed solution is indeed a valid solution.

The class of all search problems that can be solved in polynomial time is denoted P. Hence, these are the search problems for which there exists a

polynomial time algorithm that actually *finds* a valid solution if such a solution exists.

By definition, **P** is a subset of **NP**. But is it a proper subset in the sense that $\mathbf{P} \neq \mathbf{NP}$? It is strongly believed, buy yet unproven, that the two sets are not the same. The TSP is in NP but no polynomial time algorithm is known that solves it. It is strongly believed that no such algorithm exists.

Reductions

A reduction from search problem A to search problem B is a polynomial-time algorithm f that transforms any instance I of A into an instance $f(I)$ of B, together with another polynomial-time algorithm h that maps any solution S of $f(I)$ back into a solution $h(S)$ of I. If $f(I)$ has no solution then neither does I. We say that A reduces to B (notation $A \to B$) if such a reduction exists.

The implication of $A \to B$ is twofold:

1. Any efficient algorithm for B can be used to solve A efficiently.
2. Solving problem B is at least as difficult as solving problem A (up to a polynomial factor in running time).

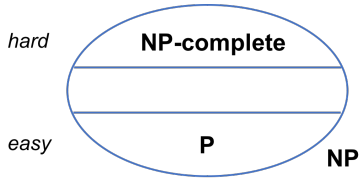
Reductions as defined above are also known as transformations: each instance of A maps to exactly one instance $f(I)$ of B and I has a solution if and only if $f(I)$ has a solution. In general, we say that A reduces to B if any algorithm for B can be used to solve A while losing only a polynomial factor in the running time. Hence, we use implication 1 above as our definition of reducibility. Note that the first implication implies the second. Hence, from now we shall use the first implication as our informal definition of a reduction.

Reductions are transitive (they compose):

$$(A \to B \text{ and } B \to C) \quad \Rightarrow \quad A \to C.$$

The class of NP-complete problems.

A search problem is called **NP**-complete if all other search problems reduce to it.
If some search problem B is **NP**-complete then, by the first implication above, any efficient algorithm for B can be used to solve *every search problem*! Note that the fact the we can define NP-complete problems does not automatically mean that such search problems exist. Nevertheless, they do and the first problem that was proven to be NP-complete (by Cook and Levin) is the Satisfiability problem (SAT). By the definition, NP-complete



problems can be seen as the hardest problems among all search problems simply because any polynomial time algorithm for any NP-complete problem can used to solve every search problem.

A problem is called NP-hard if all search problems can be reduced to it. The subtle difference with the definition of NP-complete is that we do not require that the NP-hard problem itself is a search problem. (So the weird problem of finding the 2^n-th shortest tour that was mentioned earlier is formally not an NP-complete problem. Nevertheless, it certainly is (NP)-hard!)

Using reductions

As mentioned, reductions are used in two ways.

1. If $A \to B$ then any algorithm for B can be used to solve A.
Example 1: Bipartite matching (A) can be reduced to Maximum Flow (B). See Exercise 4 of week 1. Any algorithm for MaxFlow (like Ford Fulkerson) can be used to find a maximum matching in a bipartite graph.
Example 2: Minimum s-t cut (A) reduces to Maximum Flow (B). Given maximum flow, we can do a breath-first search in the residual graph to find the set of vertices that can still be reached from the source s. This set then defines a minimum s-t cut.

P (easy)	**NP**-complete (hard)
Euler tour	Rudrata tour
Shortest path	Longest path
Chinese Postman	Traveling Salesman
Linear Programming (LP)	Integer Linear Programming (ILP)
2SAT	Satisfiability (SAT) and 3SAT
Minimum Spanning Tree	Vehicle Routing
Matching	Load Balancing
Maximum Flow	Knapsack
Minimum Cost Flow	Vertex Cover
...	...

Figure 4: Although both lists are essentially infinitely long, most problems in practice are hard. Some problems come in easy-hard pairs like Euler-Hudrata and LP-ILP.

2. If $A \to B$ and A is NP-complete then B is NP-complete as well.
This follows from the transitivity of reductions

$$\text{'Any search problem'} \to A \to B$$

$$\Rightarrow$$

$$\text{'Any search problem'} \to B.$$

Example. Cook and Levin showed by a complicated proof that SAT is NP-complete. The 3SAT problem is a special case of SAT and reducing SAT to 3SAT is relatively easy. Hence, after SAT what proven to be NP-complete, NP-completeness of 3SAT, and many more problems, followed shortly after.

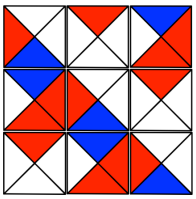


Figure 5: Solution to the tiling instance of Figure 3 (left). This structure of 3×3 can be infinitely repeated.

Exercises

Exercise 0 Exercise 0.1 from Chapter 0 of Dasgupta. You are not asked to formally prove this from the definition. Just use your intuition to tell which is true $O()$, $\Omega()$ or both (and see if you were correct from the answers provided later.)

Exercise 1 Exercise 8.1 from Dasgupta. Remember that all distance are assumed integer.

Exercise 2 Exercise 8.2 from Dasgupta.

Exercise 3 Exercise 8.10 from Dasgupta. Hint: The following problems were mentioned as NP-complete in the chapter and they provide the answer to subquestions (a) - (g) (in some order). See Chapter 8 for their definitions. Rudrata Path, Rudrata Cycle, Clique, Vertex Cover, SAT, Independent Set.

Exercise 4 Exercise 8.12 from Dasgupta.

Exercise 5 Exercise 8.13 from Dasgupta.

Exercise 6 Exercise 8.16 from Dasgupta. Hint: you do not need to know or use the definition of 3SAT to answer this question. It satisfies to know that it is an NP-complete problem.

Exercise 7 Exercise 8.21 from Dasgupta.
An example: If $x = ACTGACG$ and $k = 4$ then the multiset of all of its k-mers is $\Gamma(x) = \{ACTG, CTGA, TGAC, GACG\}$.