Types of Algorithms

CHAPTER 5: APPROXIMATION ALGORITHMS

1. Introduction

Most interesting everyday optimization problems are extremely challenging from a computational viewpoint. In fact, quite often, discovering a near-optimal or even an optimal solution to an optimization problem of large-scale may necessitate computational resources beyond what is essentially available. There is a significant body of literature discovering the computational properties of various optimization problems by seeing how the computational strains of a solution technique grow with the extent of the problem case to be solved (Aho et al., 1976; 1979; Alon & Spencer, 2000). A key distinction is made amongst problems that need computational resources that develop polynomially with problem extent versus those for which the necessary resources grow exponentially. The former class of problems is termed efficiently solvable, however, problems in the latter class are deemed *intractable* since the exponential growth in necessary computational resources reduces all but the smallest cases of such problems unsolvable (Cook & Rohe, 1999; Chazelle et al., 2001; Carlson et al., 200).

It has been concluded that a large number of common optimization problems are categorized as *NP*-hard. It is widely assumed—though not yet verified (Clay Mathematics Institute, 2003)—that NP-hard problems are obstinate, which reflects that there is not any effective algorithm (i.e. one that measures polynomially) that is guaranteed to discover an optimal solution for this type of problems. NP-hard optimization tasks' instances are the minimum bin packing problem, the minimum traveling salesman problem, and the minimum graph coloring problem. As a result of the character of NP-hard problems, an advancement that leads to a better appreciation of the computational properties, structure, and means of solving one of them, approximately or exactly, also leads to improved algorithms for resolving hundreds of other diverse but related NP-hard problems. A number of computational problems, in regions as diverse as computer-aided design and finance, operations research, economics, biology, have been revealed to be NP-hard. (Aho & Hopcrosft, 1974; Aho et al., 1979; 1991).

A natural query is whether near-optimal (i.e. *approximate*) solutions can probably be found efficiently for hard optimization problems like these. Heuristic local search techniques, such as simulated annealing and tabu search (see Chapters 6 and 7), are usually quite effective at finding *approximate* solutions. However, these techniques do not come with rigorous assurances concerning the class of the absolute solution or the requisite maximum runtime. In this chapter, we will discourse a more theoretical methodology to this issue involving alleged "approximation algorithms", which are efficient algorithms that can be verified to produce solutions of a definite quality. We will also discuss categories of problems for which no effective approximation algorithms exist, hence leaving an important part for the quite common heuristic local search methods (Shaw et al., 1998; Dotu et al., 2003).

The design of decent approximation algorithms is an extremely active area of research in which new methods and techniques are found. It is quite possible that these methods will become of increasing significance in tackling large everyday optimization problems (Feller, 1971; Hochbaum, 1996; Cormen et al., 2001).

In the early 1970s and late 1960s, a precise idea of approximation was projected in the context of bin packing and multiprocessor scheduling (Graham, 1966; Garey et al., 1972; 1976; Johnson, 1974). Approximation algorithms, in general, have two properties. First, they provide a reasonable solution to a problem case in polynomial time. In most circumstances, it is not difficult to develop a procedure that discovers some feasible solution (Kozen, 1992). Though, we are concerned with having some assured class of the solution that is the second aspect describing approximation algorithms. The class of an approximation algorithm remains the maximum "distance" between the optimal solutions and its solutions, assessed over all the possible causes of the problem. Casually, an algorithm approximately resolves an optimization problem if it continually returns a possible solution whose measure is near to optimal, for instance within a factor confined by a constant or by a gradually increasing function of the input size. Assuming constant α, an algorithm A is an α-approximation algorithm for a particular minimization problem Π if its answer is as a maximum α time the optimum, in view of all the possible cases of problem Π.

This chapter focuses on NP-hard optimization problems and their design of approximation algorithms. We will show in what way standard algorithm design methods such as local search and greedy methods have been used to develop good approximation algorithms. We will also show exactly how a randomization is a potent tool for scheming approximation algorithms. Randomized algorithms are fascinating because in general such methods are easier to implement and analyze, and quicker than deterministic algorithms (Motwani & Raghavan, 1995). A randomized algorithm is basically an algorithm that performs a few of its choices arbitrarily; it "flips a coin" to choose what to do at some phases. As a result of its random component, various executions of a randomized algorithm can result in different runtime and solutions, even when seeing the same case of a problem. We will demonstrate how one can associate randomization with approximation methods in order to approximate NP-hard optimization problems efficiently. In this case, the runtime of the approximation algorithm, the approximation solution, and the approximation ratio may be random variables. Challenged with an optimization problem, the aim is to generate a randomized approximation algorithm using runtime provably confined by a polynomial and whose possible solution is near to the optimal solution, *in probability*. Note that these guarantees stand for every case of the problem being solved. The only arbitrariness in the performance warranty of the randomized approximation algorithm derives from the algorithm itself & not from the instances.

Since we do not see efficient algorithms to discover optimal solutions for NP-hard problems, a crucial question is whether we can proficiently compute decent approximations that are near to optimal. It would be very exciting (and practical) if one could see from exponential to polynomial time intricacy by relaxing the check on optimality, especially if we assure at most a relatively small error (Qi, 1988; Vangheluwe et al., 2003; Xu, 2005).

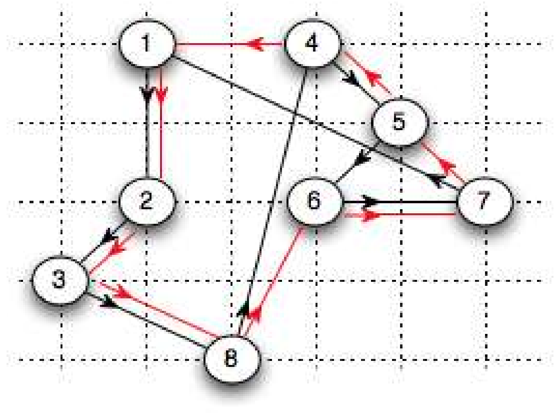

Fig. 22. Schematic Illustration of Mechanism for Approximation Algorithms

[Source: http://faculty.ycp.edu/~dbabcock/PastCourses/cs360/lectures/lecture29.html]

Decent approximation algorithms have been suggested for some significant problems in combinatorial optimization. The so-called APX intricacy class comprises the problems that permit a polynomial-time approximation algorithm by a performance ratio confined by a constant. For some problems, we can plan even better approximation algorithms. More precisely we can contemplate a group of approximation algorithms that permits us to get as near to the optimum as we want, as long as we are ready to trade quality with time (Reinelt, 1994; Indrani, 2003). This special group of algorithms is termed an approximation scheme (AS) & the so-called PTAS category is the category of optimization problems that permit on behalf of a polynomial time approximation scheme that gauges polynomially in the extent of the input. In some instances, we can devise approximation systems that gauge polynomially, both in the magnitude of the approximation error and in the size of the input. We refer to the category of problems that permit this type of fully polynomial time approximation schemes by way of FPTAS (Faigle et al., 1989; Boyd & Pulleyblank, 1990; Gomes & Shmoys, 2002).

For some NP-hard problems, however, the approximations that have been achieved so far are quite poor, & in some instances, no one has ever been capable of devising approximation algorithms in the optimum constant factor (Chen & Epley, 1970; Hochbaum & Shmoys, 1987). Initially, it was not obvious if these weak outcomes were due to our deficiency of ability in devising decent approximation algorithms for this type problems or to some intrinsic structural characteristic of the problems that disregards them from having decent approximations. We will see that actually there are restrictions to an approximation which are *inherent* to some categories of problems (Graham, 1969; Graham & Pollak, 1971; McCrary et al., 2000). For example, in some instances, there is a poorer bound on the approximation constant factor, and in other instances, we can probably demonstrate that there are not any approximations within some constant factor of the optimum. Essentially, there is an extensive range of scenarios starting from NP-hard optimization problems that permit approximations to *any* essential degree, to problems not permitting approximations at all. We will deliver a brief overview of proof techniques used to develop non-approximability results (Ryser, 1951; Andersen & Hilton, 1983; Pulleyblank, 1989).

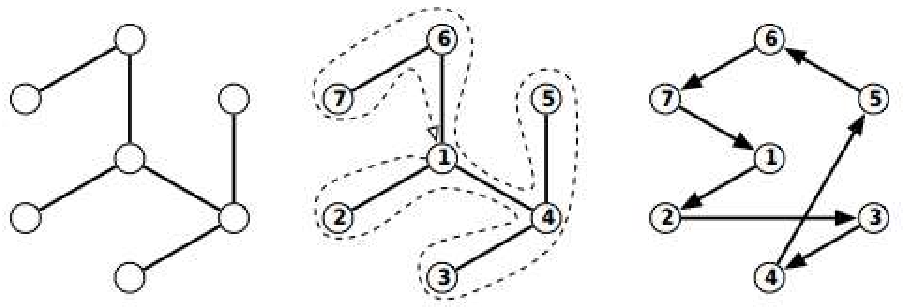

Fig. 23. Approximation route for an approximation algorithm problem

[Source: http://fliphtml5.com/czsc/chmn/basic]

We believe that the finest way to understand the notions behind randomization and approximation is to study cases of algorithms with these characteristics, through examples. Thus in each segment, we will first present the intuitive concept, then highlight its salient points through well-selected instances of prototypical problems (Banderier et al., 2003; Podsakoff et al., 2009; Bennett et al., 2015). Our aim is far from trying to deliver a comprehensive analysis of approximation algorithms or the ideal approximation algorithms for the introduced problems. Instead, we describe the various design and evaluation methods for randomized and approximation algorithms, using obvious examples that allow for comparatively simple and intuitive descriptions. For some problems discoursed in the chapter, there are approximations with enhanced performance guarantees but needing more sophisticated proof methods that are beyond the range of this introductory tutorial. In such instances, we will guide the reader to the related literature results (Spyke, 1998; Diening et al., 2004; Becchetti et al., 2006).

2. Approximation Strategies

2.1.Optimization Problems

We will describe optimization problems in an orthodox way (Aho et al., 1979; 1981; Ausiello et al., 1999). There are three defining features of each optimization problem: the criterion of a possible *solution* to the problem, the configuration of the input *instance*, & the *measure* function used to decide which possible solutions are deliberated to be optimal. It will be obvious from the problem title whether we desire a possible solution with a maximum or minimum measure. To explain, the minimum vertex cover problem can be defined in the following manner (Colbourn, 1984; Ansótegui et al., 2004; Leahu & Gomes, 2004).

Minimum Vertex Cover case is illustrated below*:*

Case: An undirected graph G = (V,E).

Solution: A subset S ⊆ V so that for every {u,v} ∈ E, either u ∈ S or v ∈ S.

Measure: |S|.

We use the following scheme for items related to a case I.

i. Sol(I) is the set of possible solutions to I;
ii. m_I: Sol(I) → R is the measure function concomitant with I;
iii. Opt(I) ⊆ Sol(I) is the possible solutions with optimal measure (either maximum or minimum).

Hence, we may completely stipulate an optimization problem Π by providing a group of tuples {(I, Sol(I), m_I, Opt(I))} over all possible cases I. It is important to remember that Sol(I) and I can be over entirely different domains. In the above case, the group of I is all pointless graphs, while Sol(I) is all possible subdivisions of vertices in a graph (Chazelle, 2000; Chazelle & Lvov, 2001; Vershynin, 2009).

2.2.Approximation and Performance

Crudely speaking, an algorithm approximately resolves an optimization problem if it returns a possible solution at all times whose measure is near to optimal. This intuition is made accurate below. Consider Π an optimization problem. We assume that an algorithm (A) *feasibly solves* Π if specified a case I ∈ Π, A(I) ∈ Sol(I); i. e, A returns a possible solution to me.

Let A feasibly resolve Π. Then we describe the *approximation ratio* α (A) of A to be the lowest possible ratio concerning the measure of A (I) & the extent of an optimal solution. Formally,

$$\alpha(A) = \min_{I \in \Pi} \frac{m_I(A(I))}{m_I(Opt(I))}$$

This ratio is each time at least 1 for minimization problems. For maximization problems, it is always at extreme 1, respectively.

2.3.Complexity Background

An optimization problem having measure 0–1 valued is defined as a decision problem. That is, solving a case I of a decision problem relates to answering a *yes/no* query about I (where *yes* relates to a measure of 1, & *no* relates to a measure of 0). We may, therefore, denote a decision problem as a subgroup S of the group of all possible cases: members of S denote instances with measure 1.

Casually, P (polynomial time) is regarded as the category of decision problems Π that have a consistent algorithm A_Π such that each instance $I \in \Pi$ is resolved by A_Π in a polynomial ($|I|^k$ for some constant k) a total number of steps on any "rational" model of computation. Rational models include single-tape & multi-tape Turing machines, pointer machines, random access machines etc (Lovász, 1975; Gurevich, 1990; Belanger & Wang, 1993).

While P is meant to signify a category of problems that can be proficiently solved, NP (non-deterministic polynomial time) is a category of decision problems Π, which can be efficiently *checked*. More correctly, NP is the category of decision problems Π that have a consistent decision problem Π^0 in P & constant k satisfying:

$$I \in \Pi \text{ if and only if there exists } C \in \{0, 1\}^{|I|k} \text{ such that } (I, C) \in \Pi'$$

In other words, one can conclude if a case I am in an NP problem that can be proficiently solved if one is also given a certain short string C that is of length polynomial in me. For instance, deliberate the NP problem of defining if a graph G having a path P that travels across all nodes exactly one time (this is called as the Hamiltonian path problem) (Johnson, 1973; Ho, 1982; Blass & Gurevich, 1990). If one is given G with an explanation of P, it is quite easy to confirm that P is certainly such a path by testing that:

i. P has all nodes in G
ii. No node seems more than one time in P
iii. Any two contiguous nodes in P have an advantage between them in G

However, it is not identified how to find this kind of path P given merely a graph G, & this is the major difference between NP and P. Actually, the Hamiltonian path problem does not only exist in NP but is present in NP-hard also, see the Introduction (Aharoni et al., 1985; Garey & Johnson, 2002).

Notice that although a short proof is always present if $I \in \Pi$, it must not be the instance that short proofs are present for instances not in Π. Therefore, while P problems are deliberated to be those that are efficiently decidable & NP problems are those deliberated to be efficiently verifiable by a short proof (Nemhauser & Ullmann, 1969; Hopper & Turton, 2001; Chazelle, 2004).

We will also contemplate the optimization counterparts to NP and P, which are NPO and PO, respectively. Informally, PO is the category of optimization problems that have a polynomial time algorithm which always yields an optimal solution to every case of the problem, while NPO is the category of optimization problems in which polynomial time computable is the measure function, and an algorithm can decide whether or not a probable solution is possible in polynomial time (Chazelle & Liu, 2001; Röglin & Vöcking, 2007; Röglin & Teng, 2009).

Here, we will focus on approximating solutions to the "toughest" of NPO problems, those problems in which the consistent decision problem is NP-hard. Amusingly, some NPO problems of this kind can be approximated very finely, whereas others can barely be approximated at all (Jiménez et al., 2001; Cueto et al., 2003; Aistleitner, 2011).

3. The Greedy Method

Greedy approximation algorithms are intended with a simple philosophy in attention: repeatedly make choices that develop one nearer and nearer to a possible solution for the problem. These choices would be optimal according to a flawed but effortlessly computable heuristic. In particular, this heuristic tends to be as opportunistic as conceivable in the short run. That is why such algorithms are termed greedy—a better name could be "short-sighted". For example, suppose my aim is to find the shortest path from my home to the theater (Klein & Young, 2010; Ausiello et al., 2012). If I assumed that the walkthrough Forbes Avenue is almost the same distance as the walkthrough Fifth Avenue, now if I am nearer to Forbes than Fifth Avenue, it would be sensible to walk towards Forbes & take that route (Wang, 1995; Khuller, 1998; Toth et al., 2017).

Obviously, the success of this strategy relies upon the correctness of my conviction that the Forbes path is definitely just as decent as the Fifth path. We will illustrate that for some problems, picking a solution conferring to an opportunistic, imperfect heuristic reaches a non-trivial approximation algorithm (Karp, 1975; Paz & Moram, 1977; Mossel et al., 2005).

3.1.Greedy Vertex Cover

In the preliminaries, the minimum vertex cover problem was described. Alternatives of the problem come up in various areas of optimization research. We will define a simple greedy algorithm, which is a 2-approximation for the problem; i. e, the vertex cover cardinality resumed by our algorithm is no larger than two times the cardinality of the least cover (Khot, 2002; Khot et al., 2007; Khot & Vishnoni, 2015). The Greedy-VC algorithm is as below.

Firstly, let S be an empty group. Choose a random edge {u,v}. Add u and v to S, & remove u & v from the graph. Repeat until no edges persist in the graph. Yield S as the vertex cover.

Proof: Firstly, we claim S as resumed by Greedy-VC is definitely a vertex cover. Suppose not; then there occurs an edge e that was not protected by any vertex in S. Since we only take out vertices from the graph which are in S, an edge e will remain in the graph once Greedy-VC had concluded, which is a contradiction (Kaufman, 1974; Liu, 1976; Durand et al., 2005).

Let S^* is a minimum vertex cover. We will now indicate that S^* contains no less than $|S|/2$ vertices. It will trail that $|S^*| \geq |S|/2$, therefore our algorithm takes a $|S|/|S^*| \leq 2$ approximation ratio.

Since the edges we picked in Greedy-VC do not share endpoints at all, it follows that:

i. S|/2 is the total number of edges we picked and
ii. S^* must have picked at least one vertex from every edge we picked.

It follows that $|S^*| \geq |S|/2$.

Occasionally when one verifies that an algorithm has a definite approximation ratio, the analysis is fairly "loose", and might not reflect the best probable ratio that can be achieved. It reflects that Greedy-VC is certainly not better than a 2-approximation. Specifically, there is an infinite group of Vertex Cover cases where Greedy-VC provably picks exactly double the number of vertices required to cover the graph, specifically in the case of comprehensive bipartite graphs (Book & Siekmann, 1986; Hermann & Pichler, 2008).

One final remark must be noted on Vertex Cover. Though the above algorithm is actually quite simple, no superior approximation algorithms are known! Actually, it is widely assumed that minimum vertex cover cannot be approximated better than $2 - \varepsilon$ for some $\varepsilon > 0$ unless P = NP (Hermann & Kolaitis, 1994; Khot & Regev, 2003).

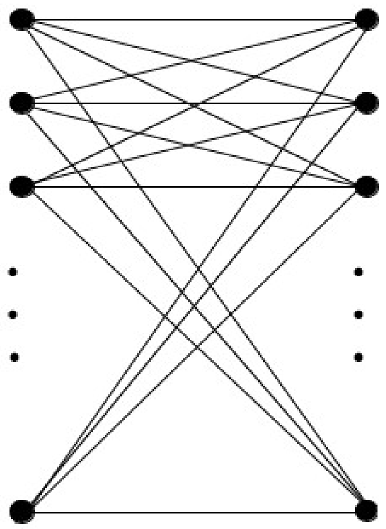

Fig. 24. a sketch of a complete bipartite graph with n nodes colored red and n nodes colored blue.

[Source: https://link.springer.com/chapter/10.1007/0-387-28356-0_18]

A graph for which its vertices can be allotted one of two colors is termed as *bipartite* (say, *blue* or *red*), in such a manner that all edges have different colored endpoints. When applying Greedy-VC on these cases (for any normal number n), the algorithm will pick all 2_n vertices.

3.2.Greedy MAX-SAT

The problem MAX-SAT has been very well-considered; variants of it arise in several areas of discrete optimization. To introduce it needs a bit of terminology. We will deal exclusively with Boolean variables (i. e, those which are either false or true), which we will represent by x_1, x_2, etc. A *literal* is explained as either negation of a variable or a variable (e.g. x_7, $\neg x_{11}$ are literals). A *clause* is explained as the OR of few literals (*e.g.* ($-x_1 \vee x_7 \vee \neg x_{11}$) is a clause). We assume that a Boolean formula exists in CNF (*conjunctive normal form*) if it is given as an AND of clauses (e.g. ($-x_1 \vee x_7 \vee -x_{11}$)$\wedge$($x_5 \vee -x_2 \vee -x_3$) is in CNF). Lastly, the MAX-SAT problem is to discover a consignment to the variables of the Boolean formula in CNF so that the maximum total of clauses are fixed to true, or are *satisfied*. Correctly:

MAX-SAT problem case is illustrated below:

Instance: A Boolean formula F in *conjunctive normal form* CNF.

Solution: An assignment a, that is a function from every variable in F to {true or false}.

Measure: The number of clauses in F which are set to true (being satisfied) while the variables in F are assigned rendering to a.

What might be a normal greedy strategy for approximately resolving MAXSAT? One approach is to choose a variable that satisfies several clauses if it is set to a definite value. Instinctively, if a variable occurs invalid in several clauses, putting the variable to *false* will gratify several clauses; hence this approach should approximately resolve the problem well. Let $n(l_i, F)$ represent the total number of clauses in F in which the literal l_i appears.

Greedy-MAXSAT: Choose a literal l_i having maximum $n(l_i, F)$ value. Set the analogous variable of l_i so that all clauses having l_i are satisfied, producing a reduced F. Repeat until no variables stay in F.

It is easy to appreciate that Greedy-MAXSAT goes in polynomial time (coarsely quadratic time, contingent with the computational model picked for analysis). It is also a "decent" approximation for the MAX-SAT problem.

3.3. Greedy MAX-CUT

Our next example illustrates how local search (specifically, *hill-climbing*) may be used in designing approximation algorithms. Hill-climbing is naturally a greedy approach: when one has a possible solution x, one tends to improve it by picking some feasible y which is "close" to x, but then has a better measure (higher or lower, depending on maximization or minimization). Repeated attempts at improvement frequently produce "locally" optimal solutions which have a good measure comparative to a universally optimal solution (i.e. a member of Opt(I)). We explain local search by proposing an approximation algorithm meant for the NP-complete MAX-CUT issue which discovers a locally optimal substantial assignment. It is important to remember that not all local search approaches try to discover a local optimum—for instance, simulated annealing tries to *escape* from local optima hoping to find a global optimum (Ghalil, 1974; Kirkpatrick et al., 1983; Černý, 1985).

MAX-CUT problem case is illustrated below*:*

Case: An undirected graph G = (V,E).

Solution: A cut of the graph, that is, a pair (S, T) such that S ⊆ V & T = V − S.

Measure: The *cut size*, that is the number of edges intersecting the cut, i.e. $|\{\{u,v\} \in E \mid u \in S, v \in T\}|$.

Our local search algorithm constantly improves the current possible solution by altering one vertex's position in the cut, till no more improvement can be attained. We will show that the cut size is at least m/2 at such a local maximum.

Local-Cut: Start with a random cut of V. For each vertex, conclude if taking it to the other part of the partition upturns the size of the cut. If so, change it. Repeat up until no such movements are probable.

First, note that this algorithm reprises at most m times, as every movement of a vertex upturns the size of the cut by no less than 1, and a cut can be as a maximum m in size.

Local-Cut is a$^{1/2}$ approximation algorithm for MAX-CUT as demonstrated below:

Proof. Suppose (S, T) be the cut yielded by the algorithm, & consider a vertex v. After the algorithm ends, observe that the total number of edges contiguous with v that cross (S, T) is more than the total number of contiguous edges that do not cross, else v would have been moved. Suppose deg(v) be the degree of (v). Then our observation suggests that however, deg(v)/2 limits out of v cross the cut yielded by the algorithm.

Let m* be the number of edges intersecting the cut returned. Each edge takes two endpoints, hence the sum/counts each intersecting edge at most twice, i.e.

$$\sum_{v \in V} (\deg(v)/2) \leq 2m^*$$

However, note $\sum_{v \in V} \deg(v) = 2m$: when adding up all degrees of vertices, each edge gets counted precisely twice, once for every endpoint. We conclude that:

$$m = \sum_{v \in V} (\deg(v)/2) \leq 2m^*$$

The algorithm has the following approximation ratio $\frac{m^*}{m} \geq \frac{1}{2}$.

It seems that MAX-CUT concedes much-improved approximation ratios than 1/2; an alleged *relaxation* of the issue to a semi-certain linear program produces a 0.8786 approximation (Goemans & Williamson, 1995). However, MAX-CUT cannot be approximated randomly as well, like several optimization problems (1 − ε, for all ε> 0) except P = NP. That is to state, it is implausible that MAX-CUT exists in the PTAS complexity class.

3.4.Greedy Knapsack

The knapsack problem & its special cases have been widely studied in operations research. The idea behind it is typical: you have a knapsack having capacity C, & a group of items 1,..., n. Each item has a certain cost c_i of carrying it, together with a profit p_i that you would gain by carrying it. The problem is then to discover a subset of objects with the cost at most C, devising maximum profit (Edmonds, 1965; Holland, 1992; Halperin, 2002).

Maximum Integer Knapsack case is illustrated below*:*

Case: A capacity $C \in N$, & a number of objects $n \in N$, with consistent costs & profits c_i, $p_i \in N$ for all $i = 1, ..., n$.

Solution: A subset $S \subseteq \{1, ..., n\}$ so that $P_{j\in S}\, c_j \leq C$.

Measure: The *total profit* $\sum_{j\in S} p_j$.

Maximum Integer Knapsack, as framed above, is NP-hard. There is also a "fractional" form of this problem (we name it Maximum Fraction Knapsack), that can be resolved in polynomial time. In this form, rather than having to choose the entire item, one is permitted to pick *fractions* of items, similar to 1/8 of the 1st item, 1/2 of the 2nd item, and so on. The corresponding profit & cost incurred from the objects will be also fractional (1/8 of the profit & cost of the 1st, 1/2 of the profit & cost of the 2nd, & so on) (Ibarra & Kim, 1975; Miller, 1976; Geman & Geman, 1987).

One greedy strategy for resolving these two problems is to box items with the biggest profit-to-cost ratio first, hoping to get several small-cost high-profit objects in the knapsack. It turns out that such algorithm will not provide any constant approximation guarantee, rather a tiny variant on this strategy will provide a 2-approximation for Integer Knapsack, & a precise algorithm for Fraction Knapsack (Adleman, 1980; Guibas et al., 1983; Lenstra et al., 1990). The algorithms for Integer Knapsack & Fraction Knapsack are, respectively:

i. Greedy-IKS: Pick items with the biggest profit-to-cost ratio first, till the total cost of items picked is greater than C. Let j be the last object is chosen, & S be the group of items picked before j. Return either {j} or S, contingent with which one is more beneficial.
ii. Greedy-FKS: Pick items as in Greedy-IKS. When the item j marks the cost of the existing solution greater than C, improve the *fraction* of j so that the resultant cost of the solution is precisely C.

We omit a proof of the succeeding. A full treatment can be seen in Ausiello et al. (1999). Greedy-KS is a 12-approximation for Maximum Integer Knapsack as illustrated below:

Proof. Fix a case of the problem. Suppose $P = \sum_{i\in S} p_i$ is the total profit of objects in S, & j be the last item picked (as specified in the algorithm). We will demonstrate that $P + p_j$ is equal to or greater than the profit of the optimal Integer Knapsack solution. It trails that one of the S or {j} has no less than half the yield (profit) of the optimal solution (Alkalai & Geer, 1996; LaForge & Turner, 2006; LaForge et al., 2006).

Suppose S_{I^*} is an optimal Integer Knapsack solution of the given case, with total profit P_{I^*}. Similarly, let S_{F^*} & P_{F^*} relate to the optimal Fraction Knapsack solution. Note that $P_{F^*} \leq P_{I^*}$. By the exploration of the algorithm aimed at Fraction Knapsack, $P_F^* = P + p_j$, in which $\in (0,1]$ is the fraction picked for object j in the algorithm. Therefore

$P + p_j \geq P + p_j = P_F^* \geq P_{I^*}$

Actually, this algorithm can be drawn-out to acquire a PTAS (*polynomial time approximation scheme*) for Maximum Integer Knapsack, (observe Ausiello et al., 1999). A PTAS has the characteristic that, for any stable $\in > 0$ provided, it yields a $(1 + \in)$-approximate solution. Added, in the input size, the runtime is polynomial, *provided that* is *constant*. This allows us to identify a runtime that possesses 1/ in the exponent. It is typical to observe a PTAS as a *group* of successively better (then also slower) approximation algorithms, individually running with a consecutively smaller $\in > 0$. This is instinctively why they are named an approximation *strategy*, as it is meant to propose that a range of algorithms are used. A PTAS is quite influential; such a scheme can approximately resolve a problem with ratios subjectively close to 1. Nevertheless, we will observe that several problems probably do not possess a PTAS, unless P = NP (Goemans & Williamson, 1995; Jain & Vazirani, 2001; Festa & Resende, 2002).

4. Sequential Algorithms

Sequential algorithms are employed for approximations on problems in which a feasible solution is a splitting of the case into subsets. A sequential algorithm "sorts" the objects of the case in some manner, and chooses partitions for the case based on this ordering (Wallace et al., 2004; Zhu & Wilhelm, 2006; Wang, 2008).

4.1.Sequential Bin Packing

We first consider the issue of Minimum Bin Packing that is similar in regard to the knapsack problems. Minimum Bin Packing case is illustrated below:

Case: A set of objects $S = \{r_1, \ldots, r_n\}$, where $r_i \in (0,1]$ for all $i = 1, \ldots, n$.

Solution: Splitting of S into bins $B_1, \ldots, B_M$ so that $\sum_{r_j \in B_i} r_j \leq 1$ for all $i = 1, \ldots, M$. *Measure*: M.

An evident algorithm for Minimum Bin Packing stays an *online* strategy. Initially, let j = 1 & have a bin B_1 available. As one runs across the input (r_1,r_2, etc), go for packing the new object r_i into the last bin employed, B_j. If r_i does not suit in B_j, make another bin B_{j+1} & put a_i in it. This algorithm is "online" since it works on the input in a stable order, and hence adding new items to the case while the algorithm is working does not alter the outcome (Herr, 1980; Smith, 1986; Stock & Watson, 2001).

Last-Bin is a 2-approximation to Minimum Bin Packing as illustrated below:

Proof. Suppose R is the sum of all objects, so $R = \sum_{r_i \in S} r^i$. Suppose m is the total number of bins employed by the algorithm, & let m_* be the lowest number of bins conceivable for the given case. Observe that $m^* \geq R$, since the total number of bins required is at least the total mass of all items (each bin embraces 1 unit). Now, given any couple of bins, B_i and B_{i+1} yielded by the algorithm, the totality of items from 'S' in B_i & B_{i+1} is at least 1; or else, we would have kept the items of B_{i+1} in B_i in its place. This indicates that $m \leq 2R$. Therefore $m \leq 2R \leq 2m^*$, & the algorithm is a 2-approximation (Price, 1973; Maurer, 1985; Berry & Howls, 2012).

An interesting workout for the reader is to build a series of examples indicating that this approximation bound, similar to the one for Greedy-VC, is constructed. As one might assume, there exist algorithms that provide better approximations than the above. For instance, we do not even deliberate the previous bins $B_1,..., B_{j-1}$ while trying to pack an a_i, just the last one is considered (Arora et al., 2001).

Motivated by this thought, consider the following alteration to Last-Bin. Choose each item a_i in declining order of size, putting a_i in the *first accessible* bin out of $B_1,..., B_j$. (So a new bin is simply created if a_i cannot be fitted in any of the former j bins.) Call this novel algorithm First-Bin. An improved approximation bound can be derived, through an elaborate analysis of cases.

4.2.Sequential Job Scheduling

One of the key issues in scheduling theory is exactly how to allocate jobs to multiple machines such that all the jobs are accomplished efficiently. Here, we will consider job accomplishment in the shortest extent of time possible. For the purposes of simplicity and abstraction, we will accept the machines are identical in dealing out power for each job. Minimum Job Scheduling is illustrated below:

Case: An integer k & a multi-set $T = \{t_1,... ,t_n\}$ of *times*, $t_i \in Q$ for all $i = 1,... ,n$ (*that is,* the t_i are fractions).

Solution: An allocation of jobs to machines, *that is,* a function a from $\{1,... ,n\}$ to $\{1,... ,k\}$.

Measure: The accomplishment time for all machines, supposing they run in parallel: $\max\{\sum_{i:a(i)=j} t_i \mid j \in \{1,... ,k\}\}$.

The algorithm we suggest for Job Scheduling is similarly online: when reading a novel job with time t_i, allocate it to the machine j which currently has the least aggregate of work; i.e., the j with minimum $\sum_{i:a(i)=j} t_i$.

Sequential Jobs is a 2-approximation meant for Minimum Job Scheduling as illustrated below:

Proof. Let j be a machine having maximum completion time, & let *i* be the catalog of the last job allocated to j by the algorithm. Let $s_{i,j}$ be the amount of all times for jobs preceding *i* that are allocated to j. (This may be assumed as the time which job *i* begins on machine j). The algorithm allocated *i* to the machine having the least extent of work, therefore all other machines j' at the moment have larger $\sum_{i:a(i)=j'} t_i$.. Hence, $s_{i,j} \leq \frac{1}{k}\sum_{i=1}^{n}$ that is, $s_{i,j}$ is less 1/k of the overall time of all jobs (remember k is the total number of machines).

Note $B = \frac{1}{k}\sum_{i=1}^{n} t_i \leq m^*$, the accomplishment time for an optimal solution, since the sum relates to the case where each machine takes exactly the equal fraction of time to complete. Hence the accomplishment time for machine j is

$S_{i,j} + t_i \leq m_* + m_* = 2m_*$

So the maximum accomplishment time is at most double that of an optimal solution. This is not the finest one can do: Minimum Job Scheduling also possesses a PTAS (Papadimitriou & Steiglitz, 1982; Vazirani, 1983).

5. Randomization

Randomness is a powerful source for algorithmic design. Upon the supposition that one has access to impartial coins that may be flipped & their values (heads or tails) taken out, a wide array of novel mathematics can be employed to support the analysis of an algorithm. It is often the instance that a simple randomized algorithm would have the same performance guarantees by means of a complicated deterministic (i.e. non-randomized) technique.

One of the most fascinating discoveries in the zone of algorithm design is that by adding randomness into a computational process may sometimes lead to a substantial speedup over purely deterministic techniques. This may be intuitively described by the subsequent set of observations. A randomized algorithm can be observed as a probability distribution upon a set of deterministic algorithms. The conduct of a randomized algorithm can fluctuate on a given input, dependent upon the random selections made by the algorithm; therefore when we contemplate a randomized algorithm, we are indirectly considering a randomly selected algorithm from a group of algorithms. If a substantial portion of these deterministic algorithms accomplishes well on the given input, at that point a strategy of resuming the randomized algorithm after a definite point in runtime will result in a speed-up (Nemhauser & Wolsey, 1988; Gomes et al., 1998).

Some randomized algorithms are capable of efficiently solving problems for which no effective deterministic algorithm is known, for example, polynomial identity testing (Motwani & Raghavan, 1995). Randomization is also a vital component in the prevalent simulated annealing method for resolving optimization problems (Kirkpatrick et al., 1983). At length, the problem of defining if a specified number is prime (a major problem in new cryptography) was only efficiently resolvable using randomization (Goldwasser & Kilian, 1986; Rabin, 1980; Solovay & Strassen, 1977). Very lately, a deterministic algorithm was discovered for primality (Agrawal et al., 2002).

5.1.Random MAX-CUT Solution

We saw earlier a greedy approach for MAXCUT that produces a 2-approximation. Using randomization, we can provide an extremely small approximation algorithm that partakes the same performance in approximation, & runs in expected polynomial time. Random-Cut: Select a random cut (i.e. a random splitting of the vertices into two groups). If there are less than m/2 edges intersecting this cut, repeat. Random-Cut remains a ½ approximation algorithm for MAX-CUT which runs in expected polynomial time as demonstrated below:

Proof. Suppose X is a random variable signifying the number of edges intersecting a cut. For i = 1,..., m, X_i will be a pointer variable which is 1 if the ith edge intersects the cut, and 0 otherwise. Then $X = \sum_{i=1}^{m} X_i$, so by linearity of m probability. $E[X] = \sum_{i=1}^{m} E[X_i]$.

Now for any edge {u, v}, the probability it intersects a randomly picked cut is 1/2. (Why? We randomly placed u & v in one of two probable partitions, so u will be in the same partition equally as v with probability 1/2.) Hence, $E[X_i] = 1/2$ for all i, so E[X] = m/2.

This only shows that by selecting a random cut, we anticipate getting at least m/2 edges intersecting. We want a randomized algorithm which always returns a good cut, & its running time is an arbitrary variable whose expectancy is polynomial. Let us calculate the probability that $X \geq m/2$ after a random cut is chosen. In the worst instance, when $X \geq m/2$ all the probability is based on m, and when $X < m/2$ all the probability is based on m/2−1. This makes the expectancy of X as high as possible, whereas making the likelihood of gaining an at least-m/2 cut small. Formally,

$m/2 = E[X] \leq (1 - \Pr[X \geq m/2])(m/2 - 1) + \Pr[X \geq m/2]m$

Resolving for $\Pr[X \geq m/2]$, it is as a minimum 2/(m+2). It follows that the estimated number of repetitions in the above algorithm is as a maximum (m+2)/2; therefore the algorithm shots in expected polynomial time, & always yields a cut of size no less than m/2.

We remark that, had we basically specified our approximation by way of "pick a random cut & stop", we would state that the algorithm goes in linear time, & has an estimated approximation ratio of 1/2.

5.2.Random MAX-SAT Solution

Earlier, we studied a greedy method for MAX-SAT which was guaranteed to gratify half of the clauses. Here we will study MAX-Ak-SAT, the limitation of MAX-SAT to CNF principles with *as a minimum* k literal per clause. Our algorithm is similar to the one for MAXCUT: *Choose an arbitrary assignment to the variables*. It is easy to indicate, using an analogous analysis to the above notion, that the estimated approximation ratio of this technique is at least $1 - 1/2^k$. More specifically, if m is the number of clauses in a formulary, the expected number of clauses gratified by an arbitrary assignment is $m - m/2^k$.

Let c be a *random* clause having k literals. The probability that every one of its literals was fixed to a value that marks them false is as a maximum $1/2^k$ since there is a possibility of 1/2 for each literal & there are as a minimum k of them. Thus the probability that c is gratified is at least $1-1/2^k$. Using a linearity of probability argument (such as in the MAX-CUT analysis) we conclude that as a minimum $m - m/2^k$ clauses are estimated to be satisfied.

6. A Tour of Approximation Classes

We will now take a stride back from our algorithmic debates, and concisely define a few of the common intricacy classes linked with NP optimization problems.

6.1. PTAS and FPTAS

PTAS and FPTAS are categories of optimization problems that few believe are nearer to the proper description of what is efficiently solvable, instead of merely P. This is for the reason that problems in these two classes can be approximated with constant ratios *subjectively close* to 1. However, with *PTAS*, while the approximation ratio gets nearer to 1, the runtime of the analogous approximation algorithm may increase exponentially with the ratio.

More formally, PTAS is the category of NPO problems Π which have an *approximation scheme*. That is, assumed $\varepsilon > 0$, there is a polynomial time algorithm A so that

i. If Π is a maximization issue, A is a$(1 + \varepsilon)$ approximation, that is, the ratio reaches 1 from the right.
ii. If Π is a minimization issue, it is a$(1 - \varepsilon\ )$ approximation (the ratio reaches 1 from the left).

As we mentioned, one disadvantage of a PTAS is that the algorithm $(1 + \varepsilon)$ could be exponential in 1/. The class FPTAS is basically PTAS but with the extra condition that the runtime is polynomial in n & 1/for the approximation algorithm.

6.2.A Few Known Results for PTAS and FPTAS

It is known that few NP-hard optimization problems can't be approximated subjectively well unless P = NP. One instance is a problem we observed at earlier, Minimum Bin Packing. This is a rare instance in which there is a modest proof that unless P = NP, the problem is not approximable.

Minimum Bin Packing is not in PTAS unless P = NP. In fact, there is no $3/2 - \varepsilon$ approximation for any $\varepsilon > 0$, unless P = NP:

To prove the outcome, we employ a reduction as of the Set Partition decision problem. Set Partitioning asks if an assumed set of natural numbers could be split into two sets which have an equal sum.

Set Partition:

Case: A multi-set $S = \{r_1,\ldots ,r_n\}$, in which $r_i \in N$ for all i =

1,... ,n.

Solution: A splitting of S into sets S_1 & S_2; *i.e.* $S_1 \cup S_2 = S$ & $S_1 \cap S_2 = \emptyset$.

Measure: m(S) = 1 if $\sum_{ri \in S1} r_i = \sum\sum_{ri \in S2} r_j$, & m(S) = 0 otherwise.

Proof. Suppose S = {r_1,... ,r_n} is a Set Partition case. Decrease to Minimum Bin Packing by letting $C = \frac{1}{2}\sum_{j=1}^{n} s_j$ (half the sum of elements in S), & considering a bin packing case of *items* $S' = \{r_1/C,... ,r_n/C\}$.

If S can be divided into two sets of the identical sum, then the minimum quantity of bins necessary for the analogous S' is 2. Conversely, if S cannot be divided in this manner, the minimum number of bins required for S^0 is at least 3, since every possible partitioning produces a set with a total greater than C. Hence, if there existed a poly-time (3/2 −ε)-approximation algorithm A, it might be used to resolve Set Partition:

i. If A (given S & C) yields a solution using as a maximum (3/2− ε)2 = 3−2 bins, then there is a Set Partition for S.
ii. If A yields a solution using as a minimum (3/2 − ε)3 = 9/2 − 3 = 4.5 − 3 bins, then there isn't any Set Partition for S.

However for any $\varepsilon \in (0, 3/2)$, 3 − 2 < 4.5 − 3

Consequently, this polynomial time algorithm differentiates between that S that may be partitioned & those that cannot, hence P = NP.

A similar result holds for issues such as MAX-CUT, MAX-SAT, & Minimum Vertex Cover. However, unlike the outcome for Bin Packing, the evidence for these appear to need the outline of probabilistically checkable proofs.

6.3.APX

APX is a (presumably) larger category than PTAS; the approximation promises for problems in it are severely weaker. An NP optimization issue Π is in APX only if there exists a polynomial time algorithm A & constant c so that A stays a c-approximation to Π.

6.4.A Few Known Results for APX

It is easy to observe that PTAS ⊆ APX ⊆ NPO. When one sees new intricacy classes & their inclusions, one of the primary questions to be requested is: How probable is it that these inclusions might be made into equalities? Unluckily, it is highly unlikely. The following relationship can be revealed between the three approximation categories we have seen.

We can assume that PTAS = APX⇐⇒ APX =NPO ⇐⇒ P = NP. Thus, if all NP optimization problems can be approximated inside a constant factor, at that point P = NP. Further, if all problems which have constant approximations may be subjectively approximated, still P = NP. Another way of putting this is: if NP problems are difficult to solve, then few of them are difficult to approximate as well. Moreover, there is a "hierarchy" of successively difficult-to-approximate problems.

One of the directions specified follows from a theorem of the prior section: earlier, we observed a constant factor approximation for Minimum Bin Packing. However, it does not possess a *PTAS* unless P = NP. This shows the course PTAS = APX ⇒ P = NP. One example of a problem which cannot be in APX until P = NP is the well-identified Minimum Traveling Salesman problem.Minimum Traveling Salesman is described below:

Case: A set $C = \{c_1, \ldots, c_n\}$ of *cities*, & a distance function d :

$C \times C \rightarrow N$.

Solution: A path through the cities, that is, a permutation $\pi : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$.

Measure: The cost of visiting cities relating to the path, i.e.

$$\sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)})$$

It is important to observe that when the spaces in the problem instances constantly obey a Euclidean metric then Minimum Traveling Salesperson possess a *PTAS* (Arora, 1998). Thus, we can say that it is the simplification of possible distances in the aforementioned problem that makes it hard to approximate. This is often the issue with approximability: a small limitation on an inapproximable problem may suddenly make it a highly approximable one.

7. Brief Introduction to PCPs

In the 1990s, the effort in probabilistically checkable proofs (PCPs) remained *the* major breakthrough in demonstrating hardness results, and possibly in theoretical computer science altogether. In essence, PCPs simply look at a little bit of a proposed proof, by randomness, but manage to arrest all of NP. As the number of bits checked by them is so small (a constant), while an efficient PCP occurs for a given problem, it infers the difficulty of *approximately solving* the similar problem as well, inside some constant factor.

The notion of a PCP ascended from a series of contemplations on proof-checking via randomness. We know NP signifies the class of problems which have "short proofs" we can prove effective. As long as NP is concerned, entirely all of the verification completed is deterministic. When a proof is incorrect or correct, a polynomial time verifier replies "yes" or "no" with 100% sureness.

However, what ensues when we relax the idea of total correctness to involve probability? Suppose we allow the proof verifier to toss impartial coins, & have a *one-sided error*. To be exact, now a randomized verifier only agrees to a correct proof having probability at least 1/2, yet still rejects any unfitting proof it reads. (We call it a *probabilistically checkable proof system*, that is, a PCP.) This slight alteration of what it means to substantiate a proof leads to an incredible characterization of NP: all of the NP decision problems may be verified by a PCP from the above type, which only tosses O(log n) coins & only checks a *constant* (O(1)) figure of bits of any particular proof! The result involves the production of highly complex error-correcting codes. We shall not debate it on a formal level now but will cite the aforementioned in the notation of a theorem.

8. Promising Application Areas for Approximation and Randomized Algorithms

8.1. Randomized Backtracking and Backdoors

Backtracking is one of the first and most natural methods employed for solving combinatorial problems. Generally, backtracking deterministically may take exponential time. Recent work has established that many real-world problems could be solved quite rapidly, once the selections made in backtracking are randomized. Particularly, problems in practice have a tendency to have minor substructures within them. These substructures have the tendency that once they are solved appropriately, the entire problem can be solved. The presence of these so-called "backdoors" (Williams et al., 2003) to problems mark them very rational to a solution using randomization. Coarsely speaking, search heuristics will mark the backdoor substructure first in the search, with a substantial probability. Therefore, by repeatedly resuming the backtracking mechanism after a definite (polynomial) length of time, the total runtime that backtracking requires discovering a solution is decreased tremendously.

8.2. Approximations to Guide Complete Backtrack Search

A promising method for solving combinatorial problems by complete (exact) methods draws on latest results on some of the finest approximation algorithms centered on linear programming (LP) relaxations (Chvatal, 1979; 1983, Dantzig, 2016) & so-called randomized rounding methods, as well as on outcomes that revealed the extreme inconsistency or "unpredictability" in the complete search procedures' runtime, often explained by professed heavy-tailed cost distributions (Gomes et al., 2000). Gomes and Shmoys (2002) suggest a *complete* randomized backtrack search technique that tightly combines constraint satisfaction problem (CSP) propagation methods with randomized LP-based approximations (Shmoys, 1995). They use as a standard domain a virtuously combinatorial problem, the quasi-group (or Latin square) completion problem (QCP). Each instance involves an n by n matrix having n^2 cells. A complete quasi-group contains a coloring of each cell using one of n colors such that there is no repetitive color in any column or row. Given an incomplete coloring of the 'n' through n cells, defining whether there is a valid accomplishment into a full quasi-group in an NP-complete problem (Colbourn, 1984). The underlying structure of this standard is similar to that originated in a series of practical applications, such as fiber optics routing, experimental design, and timetabling problems (Laywine & Mullen, 1998; Kumar et al., 1999).

Gomes and Shmoys compare their outcomes for the hybrid CSP/LP strategy steered through the LP randomized rounding approximation using a CSP strategy & with ann LP strategy. The results indicate that the hybrid approach considerably improves over the pure approaches on hard instances. This proposes that LP randomized rounding approximation gives powerful heuristic regulation to the CSP search.

8.3.Average Case Complexity and Approximation

While “worst case" complexity partakes a very rich theory, it frequently feels too restrictive to be pertinent to practice. Maybe NP-hard problems are hard just for some esoteric sets of cases that will hardly ever ascend. To this end, researchers have suggested theories of “average case" complexity, that attempt to probabilistically explore problems based on randomly selected instances over distributions; for an overview to this line of work, cf. (Gurevich, 1991; Nowakowski & Skarbek, 2006). Lately, an exciting thread of theoretical research has explained the connections between the average-instance complexity of problems & their approximation hardness (Feige, 2002; Wilkinson, 2003; Beier et al., 2007). For instance, it is presented that if *random 3-SAT* is difficult to solve in polynomial time (given reasonable definitions of “random” & “hard”), then NP-hard optimization problems, for example, Minimum Bisection is difficult to approximate in the worst instance. Conversely, this implies that better approximation algorithms for some problems might lead to the average-instance tractability of others. A natural research query is: does a PTAS suggest average-instance tractability or vice versa? We suspect that some proclamation of this form might be the instance. In our defense, the latest paper illustrates that *Random* Maximum Integer Knapsack is precisely solvable in expected polynomial time (Beier & Vocking, 2003; 2004; 2006).

9. Tricks of the Trade

One major initial incentive for the learning of approximation algorithms was to deliver a new theoretical avenue for coping and analyzing with hard problems. Faced with a brand-new fascinating optimization problem, how could one apply the techniques deliberated here? One possible scheme continues as follows:

i. First, try to substantiate your problem is NP-hard, otherwise, find proof that it is not! Possibly the problem admits an exciting exact algorithm, without the requirement for approximation.

ii. Often, a very intuitive and natural idea is the base of an approximation algorithm. How good is a randomly picked possible solution for the problem? (What is the anticipated value of a random solution?) What about a greedy strategy? Can you define a region such that local search does fine?

iii. Seek for a problem (name it Π) that is similar to yours in some sense, & use a present approximation algorithm for Π to get an approximation for your problem.

iv. Try to ascertain it cannot be approximated finely, by reducing few hard-to-approximate problems to your problem.

The first, third, & fourth points essentially pivot on one's resourcefulness: one's persistence to scour the literature (& colleagues) for problems related to the one at hand, in addition to one's ability to see the relationships & reductions which indicate that a problem is indeed analogous.

This chapter has been mostly concerned with the second point. To answer the queries of that point, it is critical to proving *limits* on optimal solutions, regarding feasible solutions that one's methods obtain. Regarding minimization (maximization) problems, one will have to prove *lower limits* (respectively, *upper limits*) on some optimal resolution for the problem. Devising lower (or upper) limits can simplify the proof greatly: one only needs to indicate that an algorithm yields a solution with value as a maximum c time the lower *limits* to indicate that the algorithm is a c-approximation.

We have proven upper & lower bounds repeatedly (explicitly or implicitly) in our verifications for approximation algorithms during this chapter—it may be informative for the reader to analyze each approximation proof & discover where we have done it. For instance, the greedy vertex cover algorithm (for choosing a maximal matching) works for the reason that even an optimal vertex cover secures, as a minimum, one of the vertices in each verge of the matching. The number of edges in a matching is a lower *limit* on the total number of nodes in an optimal vertex cover, and hence the total nodes in the matching (that is twofold the number of edges) are, as a maximum, twofold the number of nodes in an optimal cover.