8 Combinatorial Optimization

So far the kinds of problems we've tackled are very general: any arbitrary search space. We've seen spaces in the forms of permutations of variables (fixed-length vectors); spaces that have reasonable distance metrics defined for them; and even spaces of trees or sets of rules.

One particular kind of space deserves special consideration. A **combinatorial optimization problem**[125] is one in which the solution consists of a combination of unique **components** selected from a typically finite, and often small, set. The objective is to find the optimal combination of components.

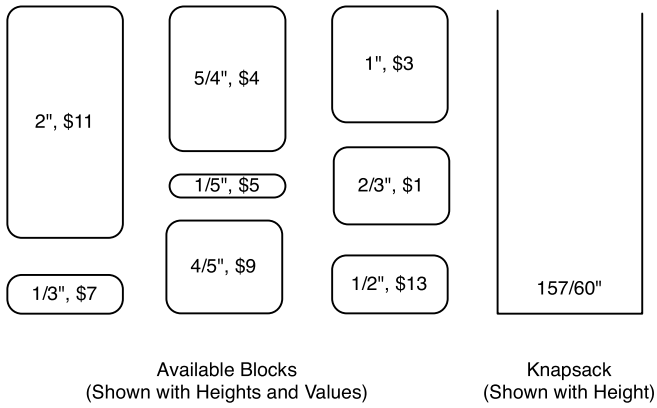

Figure 54 A knapsack problem. Fill the knapsack with as much value ($$$) without exceeding the knapsack's height.

A classic combinatorial optimization problem is a simple form of the **knapsack problem**: we're given *n blocks* of different heights and worth different amounts of money (unrelated to the heights) and a *knapsack*[126] of a certain larger height, as shown in Figure 54. The objective is to fill the knapsack with blocks worth the most $$$ (or €€€ or ¥¥¥) without overfilling the knapsack.[127] Blocks are the components. Figure 55 shows various combinations of blocks in the knapsack. As you can see, just because the knapsack is maximally filled doesn't mean it's optimal: what counts is how much value can be packed into the knapsack without going over. Overfull solutions are **infeasible** (or *illegal* or *invalid*).

This isn't a trivial or obscure problem. It's got a lot of literature behind it. And lots of real-world problems can be cast into this framework: knapsack problems show up in the processor queues of operating systems; in allocations of delivery trucks along routes; and in determining how to get exactly $15.05 worth of appetizers in a restaurant.[128]

Another example is the classic **traveling salesman problem** (or **TSP**), which has a set of *cities* with some number of *routes* (plane flights, say) between various pairs cities. Each route has a *cost*. The salesman must construct a *tour* starting at city A, visiting all the cities at least once, and finally

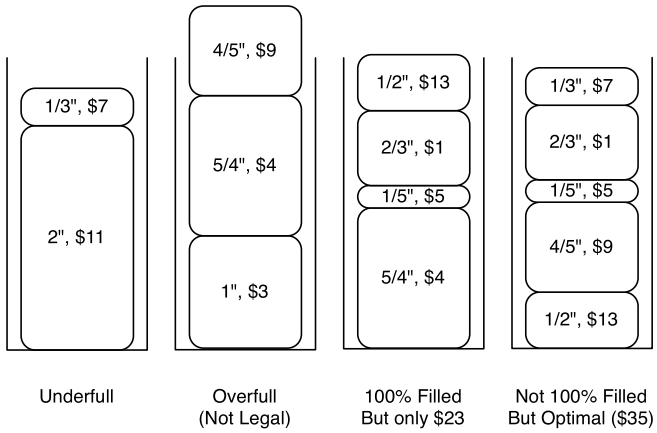

Figure 55 Filling the knapsack.

[125] Not to be confused with *combinatorics*, an overall field of problems which could reasonably include, as a small subset, practically everything discussed so far.

[126] Related are various **bin packing** problems, where the objective is to figure out how to arrange blocks so that they will fit correctly in a multi-dimensional bin.

[127] There are various knapsack problems. For example, another version allows you to have as many copies of a given block size as you need.

[128] http://xkcd.com/287/

returning to A. Crucially, this tour must have the lowest cost possible. Put another way, the cities are nodes and the routes are edges in a graph, labelled by cost, and the object is to find a minimum-cost cycle which visits every node at least once. Here the components aren't blocks but are rather the edges in the graph. And the arrangement of these edges matters: there are lots of sets of edges which are nonsense because they don't form a cycle.

Costs and Values While the TSP has **cost** (the edge weights) which must be minimized, Knapsack instead has **value** ($$$) which must be maximized. These are really just the same thing: simply negate or invert the costs to create values. Most combinatorial optimization algorithms traditionally assume costs, but we'll include both cases. At any rate, one of many ways you might convert the cost of a component C_i into a value (or vice versa) would be something along the lines of:

$$\text{Value}(C_i) = \frac{1}{\text{Cost}(C_i)}$$

That's the relationship we'll assume in this Section. This of course assumes that your costs (and values) are > 0, which is the usual case. If your costs or values are both positive and negative, some of the upcoming methods do a kind value-proportional selection, so you'll need to add some amount to make them all positive. Finally, there exist problems in which components all have exactly the same value or cost. Or perhaps you might be able to provide your algorithm with a **heuristic**[129] that you as a user have designed to favor certain components over others. In this case you could use $\text{Value}(C_i) = \text{Heuristic}(C_i)$.

Knapsack does have one thing the TSP doesn't have: it has additional **weights**[130] (the block heights) and a maximum "weight" which *must not* be exceeded. The TSP has a different notion of infeasible solutions than simply ones which exceed a certain bound.

8.1 General-Purpose Optimization and Hard Constraints

Combinatorial optimization problems can be solved by most general-purpose metaheuristics such as those we've seen so far, and in fact certain techniques (Iterated Local Search, Tabu Search, etc.) are commonly promoted as combinatorics problem methods. But some care must be taken because most metaheuristics are really designed to search much more general, wide-open spaces than the constrained ones found in most combinatorial optimization problems. We can adapt them but need to take into consideration these restrictions special to these kinds of problems.[131]

As an example, consider the use of a boolean vector in combination with a metaheuristic such as simulated annealing or the genetic algorithm. Each slot in the vector represents a component, and if the slot is true, then the component is used in the candidate solution. For example, in Figure 54 we have blocks of height 2, $\frac{1}{3}$, $\frac{5}{4}$, $\frac{1}{5}$, $\frac{4}{5}$, 1, $\frac{2}{3}$, and $\frac{1}{2}$. A candidate solution to the problem in this Figure would be a vector of eight slots. The optimal answer shown in Figure 55 would be $\langle$false, true, false, true, true, false, true, true$\rangle$, representing the blocks $\frac{1}{3}$, $\frac{1}{5}$, $\frac{4}{5}$, $\frac{2}{3}$, and $\frac{1}{2}$.

The problem with this approach is that it's easy to create solutions which are infeasible. In the knapsack problem we have declared that solutions which are larger than the knapsack are

[129] A heuristic is a rule of thumb provided by you to the algorithm. It can often be wrong, but is right often enough that it's useful as a guide.

[130] Yeah, confusing. TSP edge weights vs. combinatorial component weights. That's just the terminology, sorry.

[131] A good overview article on the topic, by two greats in the field, is Zbigniew Michalewicz and Marc Schoenauer, 1996, Evolutionary algorithms for constrained parameter optimization problems, *Evolutionary Computation*, 4(1), 1–32.

simply illegal. In Knapsack, it's not a disaster to have candidate solutions like that, as long as the final solution is feasible — we could just declare the quality of such infeasible solutions to be their distance from the optimum (in this case perhaps how overfull the knapsack is). We might punish them further for being infeasible. But in a problem like the Traveling Salesman Problem, our boolean vector might consist of one slot per edge in the TSP graph. It's easy to create infeasible solutions for the TSP which are simply nonsense: how do we assess the "quality" of a candidate solution whose TSP solution isn't even a tour?

The issue here is that these kind of problems, as configured, have **hard constraints**: there are large regions in the search space which are simply invalid. Ultimately we want a solution which is feasible; and during the search process it'd be nice to have feasible candidate solutions so we can actually think of a way to assign them quality assessments! There are two parts to this: initialization (construction) of a candidate solution from scratch, and Tweaking a candidate solution into a new one.

Construction Iterative construction of components within hard constraints is sometimes straightforward and sometimes not. Often it's done like this:

1. Choose a component. For example, in the TSP, pick an edge between two cities A and B. In Knapsack, it's an initial block. Let our current (partial) solution start with just that component.
2. Identify the subset of components that can be concatenated to components in our partial solution. In the TSP, this might be the set of all edges going out of A or B. In Knapsack, this is all blocks that can still be added into the knapsack without going over.
3. Tend to discard the less desirable components. In the TSP, we might emphasize edges that are going to cities we've not visited yet if possible.
4. Add to the partial solution a component chosen from among those components not yet discarded.
5. Quit when there are no components left to add. Else go to step 2.

This is an intentionally vague description because iterative construction is almost always highly problem-specific and often requires a lot of thought.

Tweaking The Tweak operator can be even harder to do right, because in the solution space feasible solutions may be surrounded on all sides by infeasible ones. Four common approaches:

- Invent a **closed** Tweak operator which automatically creates feasible children. This can be a challenge to do, particularly if you're including crossover. And if you create a closed operator, can it generate all possible feasible children? Is there a bias? Do you know what it is?
- Repeatedly try various Tweaks until you create a child which is feasible. This is relatively easy to do, but it may be computationally expensive.
- Allow infeasible solutions but construct a quality assessment function for them based on their distance to the nearest feasible solution or to the optimum. This is easier to do for some problems than others. For example, in the Knapsack problem it's easy: the quality of an overfull solution could be simply based on how overfull it is (just like underfull solutions).

- Assign infeasible solutions a poor quality. This essentially eliminates them from the population; but of course it makes your effective population size that much smaller. It has another problem too: moving *just* over the edge between the feasible and infeasible regions in the space results in a huge decrease in quality: it's a Hamming Cliff (see Representation, Section 4). In Knapsack, for example, the best solutions are very close to infeasible ones because they're close to filled. So one little mutation near the best solutions and whammo, you're infeasible and have big quality punishment. This makes optimizing near the best solutions a bit like walking on a tightrope.

None of these is particularly inviting. While it's often easy to create a valid construction operator, making a good Tweak operator that's closed can be pretty hard. And the other methods are expensive or allow infeasible solutions in your population.

Component-Oriented Methods The rest of this Section concerns itself with methods specially designed for certain kinds of spaces often found in combinatorial optimization, by taking advantage of the fact that the solutions in these spaces consist of *combinations of components* drawn from a typically *fixed set.* It's the presence of this fixed set that we can take advantage of in a greedy, local fashion by maintaining historical "quality" values, so to speak, of individual components rather than (or in addition to) complete solutions. There are two reasons you might want to do this:

- While constructing, to tend to select from components which have proven to be better choices.
- While Tweaking, to modify those components which appear to be getting us in a local optimum.

We'll begin with a straightforward metaheuristic called **Greedy Randomized Adaptive Search Procedures** (or **GRASP**) which embodies the basic notion of constructing combinatorial solutions out of components, then Tweaking them. From there we will move to a related technique, **Ant Colony Optimization**, which assigns "historical quality" values to these components to more aggressively construct solutions from the historically "better" components. Finally, we'll examine a variation of Tabu Search called **Guided Local Search** which focuses instead on the Tweak side of things: it's designed to temporarily "punish" those components which have gotten the algorithm into a rut.

Some of these methods take advantage of the "historical quality" values of individual components, but use them in quite different ways. Ant Colony Optimization tries to favor the best-performing components; but Guided Local Search gathers this information to determine which low-performing components appear to show up often in local optima.

The meaning of Quality or Fitness Because combinatorial problems can be cast as either cost or as value, the meaning of *quality* or *fitness* of a candidate solution is shaky. If your problem is in terms of *value* (such as Knapsack), it's easy to define quality or fitness simply as the sum total value, that is, $\sum_i \text{Value}(C_i)$, of all the components C_i which appear in the candidate solution. If your problem is in terms of *cost* (such as the TSP), it's not so easy: you want the presence of many low-cost components to collectively result in a high-quality solution. A common approach is to define quality or fitness as $1/(\sum_i \text{Cost}(C_i))$, for each component C_i that appears in the solution.

8.2 Greedy Randomized Adaptive Search Procedures

At any rate, let's start easy with a single-state metaheuristic which is built on the notions of constructing and Tweaking feasible solutions, but which doesn't use any notion of component-level "historical quality": **Greedy Randomized Adaptive Search Procedures** or **GRASP**, by Thomas Feo and Mauricio Resende.[132] The overall algorithm is really simple: we create a feasible solution by constructing from among highest value (lowest cost) components (basically using the approach outlined earlier) and then do some hill-climbing on the solution.

Algorithm 108 *Greedy Randomized Adaptive Search Procedures (GRASP)*

$C \leftarrow \{C_1, ..., C_n\}$ components
$p \leftarrow$ percentage of components to include each iteration
$m \leftarrow$ length of time to do hill-climbing

$Best \leftarrow \square$
repeat
    $S \leftarrow \{\}$ ▷ Our candidate solution
    repeat
        $C' \leftarrow$ components in $C - S$ which could be added to S without being infeasible
        if C' is empty **then**
            $S \leftarrow \{\}$ ▷ Try again
        else
            $C'' \leftarrow$ the p% highest value (or lowest cost) components in C'
            $S \leftarrow S \cup \{$component chosen uniformly at random from $C''\}$
    until S is a complete solution
    for m times **do**
        $R \leftarrow$ Tweak(Copy(S)) ▷ Tweak must be closed, that is, it must create feasible solutions
        if Quality(R) $>$ Quality(S) **then**
            $S \leftarrow R$
    if $Best = \square$ or Quality(S) $>$ Quality($Best$) **then**
        $Best \leftarrow S$
until $Best$ is the ideal solution or we have run out of time
return $Best$

Instead of picking the p% best available components, some versions of GRASP pick components from among the components whose value is no less than (or cost is no higher than) some amount. GRASP is more or less using a truncation selection among components to do its initial construction of candidate solutions. You could do something else like a tournament selection among the components, or a fitness-proportionate selection procedure (see Section 3 for these methods).

GRASP illustrates one way how to construct candidate solutions by iteratively picking components. But it's still got the same conundrum that faces evolutionary computation when it comes to the Tweak step: you have to come up with some way of guaranteeing closure.

[132] The first GRASP paper was Thomas A. Feo and Mauricio G. C. Resende, 1989, A probabilistic heuristic for a computationally difficult set covering problem, *Operations Research Letters*, 8, 67–71. Many of Resende's current publications on GRASP may be found at http://www.research.att.com/~mgcr/doc/