Combinatorial optimization

Eugeniy E. Mikhailov

The College of William & Mary

Lecture 17

Combinatorial optimization problem statement

We still want to optimize (minimize) our multi dimensional merit function E

Find $\vec{x}$ that minimizes $E(\vec{x})$ subject to $g(\vec{x}) = 0, h(\vec{x}) \leq 0$

The only difference is that values of the $\vec{x}$ are discrete, i.e., any component of the $\vec{x}$ can take a **countable** set of different values.

In this case, we cannot run our golden search algorithm or anything else which assumes continuous space for the $\vec{x}$.

Instead, we have to find a method to search through discrete sets of all possible input values, i.e. go through all possible combinations of $\vec{x}$ components.

Hence, the name combinatorial optimization.

Example: Backpack problem

Suppose you have a backpack with a given size (volume). You have a set of objects with given volumes and values (for example their cost).

Our job is to find a such subset of items that still fits in the backpack and has the maximum combined value.

For simplicity, we will assume that every item occurs only once. Then our job is to maximize

$$E(\vec{x}) = \sum \text{value}_i x_i = \overrightarrow{\text{values}} \cdot \vec{x}$$

Subject to the following constrains

$$\sum \text{volume}_i x_i = \overrightarrow{\text{volumes}} \cdot \vec{x} \leq \text{BackpackSize}$$

Where $x_i = (0 \text{ or } 1)$, i.e., it reflects whether we take this object or not

Brute force optimization

With this approach, we will just try all possible combinations of items and find the best of them.

Notice that if there are N objects, we have 2^N of all possible combinations to choose from.

So the size of the problem space and, thus, the solving time grows exponentially.

Backpack optimization: new test set generation

Recall that we are looking for an optimal direction among all possible $\vec{x}$
Generally $\vec{x}$ is a combination of zeros and ones
$\vec{x} = [0, 1, 0, 1, \cdots, 1, 1, 0, 1, 1]$
How would we generate all possible combinations of $\vec{x}$ components?

Backpack optimization: new test set generation

Recall that we are looking for an optimal direction among all possible $\vec{x}$
Generally $\vec{x}$ is a combination of zeros and ones
$\vec{x} = [0, 1, 0, 1, \cdots, 1, 1, 0, 1, 1]$
How would we generate all possible combinations of $\vec{x}$ components?

- $\vec{x}$ looks like a binary number.
- let's start with $\vec{x} = [0, 0, 0, 0, \cdots, 0, 0]$
- every new component will be generated by adding 1 to the previous x according to binary addition rules
  - for example
    $x_{next} = [1, 0, 1, \cdots, 1, 1, 0, 1, 1] + 1 = [1, 0, 1, \cdots, 1, 1, 1, 0, 0]$
- for every new $\vec{x}$, we check to see if the items fit into the backpack and if new fitted value is larger then the previous
- once we have tried all 2^N combinations of $\vec{x}$, we are done

The time of the optimization grows exponentially with the number N of items to chose, but we will find the global optimum.

Backpack optimization: test run

For realization of this algorithm, have a look at the
`backpack_binary.m`
Sample run

```
backpack_size=7;
volumes=[  2,  5,  1,  3, 3];
values =[ 10, 12, 23, 45, 4];
[pbest, max_fitted_value] = ...
  backpack_binary( backpack_size, volumes, values)

pbest = [1 3 4]
max_fitted_value = 78
```

Backpack optimization: test run

For realization of this algorithm, have a look at the `backpack_binary.m`

Sample run

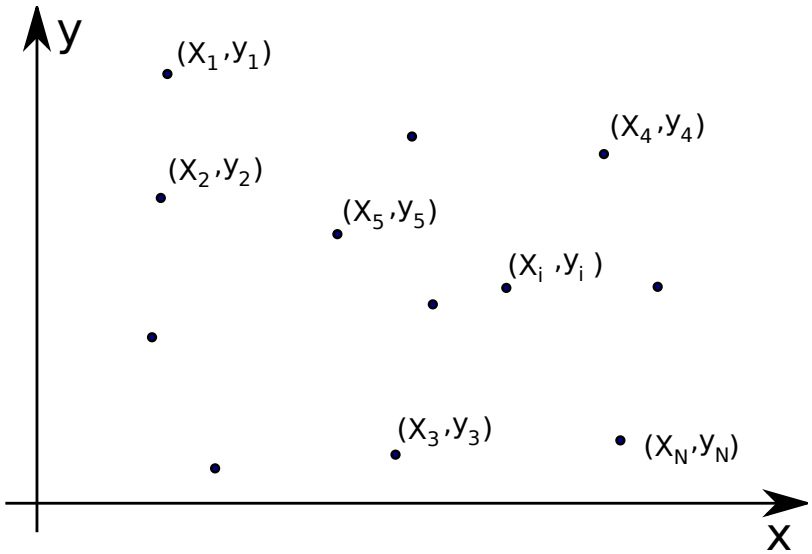
```
backpack_size=7;
volumes=[  2,  5,  1,  3, 3];
values =[ 10, 12, 23, 45, 4];
[pbest, max_fitted_value] = ...
  backpack_binary( backpack_size, volumes, values)

pbest = [1 3 4]
max_fitted_value = 78
```

- My computer sorts 20 items in 47 seconds, but 30 items would take 1000 times longer something like 13 hours to solve.

Example: Traveling salesman problem

- Suppose that you have N cities (with given coordinates) to visit
- A salesman starts in the city 1 and need to be in the city *N* at the end of a route
- Find the shortest route, so the salesman visits every city only once

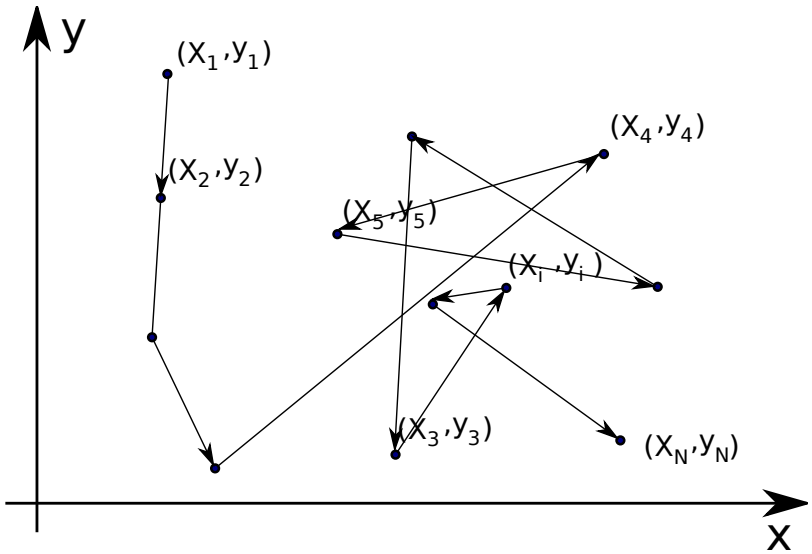
This problem has a lot of connections to the real world. Every time you ask your GPS to find a route, the GPS unit has to solve this problem. Layout of traces on a printed circuit board is essentially the same problem, as well.



Example: Traveling salesman problem

- Suppose that you have N cities (with given coordinates) to visit
- A salesman starts in the city 1 and need to be in the city *N* at the end of a route
- Find the shortest route, so the salesman visits every city only once

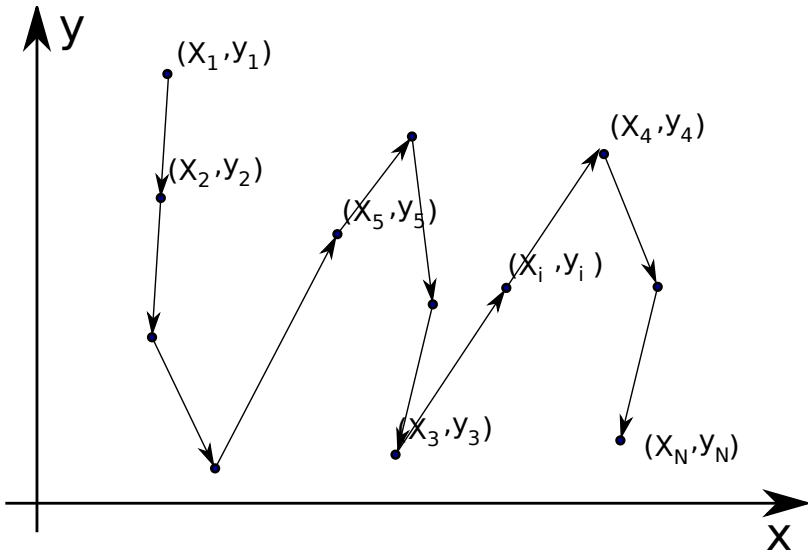
This problem has a lot of connections to the real world. Every time you ask your GPS to find a route, the GPS unit has to solve this problem. Layout of traces on a printed circuit board is essentially the same problem, as well.



Example: Traveling salesman problem

- Suppose that you have N cities (with given coordinates) to visit
- A salesman starts in the city 1 and need to be in the city *N* at the end of a route
- Find the shortest route, so the salesman visits every city only once

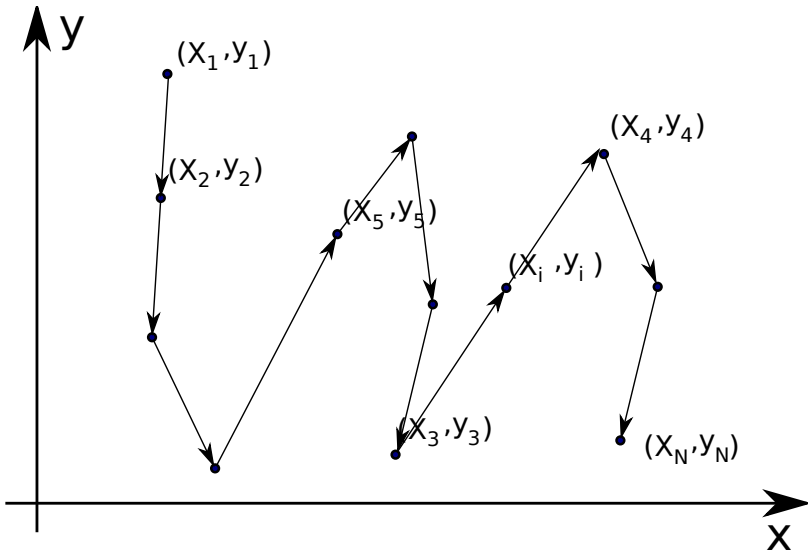
This problem has a lot of connections to the real world. Every time you ask your GPS to find a route, the GPS unit has to solve this problem. Layout of traces on a printed circuit board is essentially the same problem, as well.



Example: Traveling salesman problem

- Suppose that you have N cities (with given coordinates) to visit
- A salesman starts in the city 1 and need to be in the city N at the end of a route
- Find the shortest route, so the salesman visits every city only once

This problem has a lot of connections to the real world. Every time you ask your GPS to find a route, the GPS unit has to solve this problem. Layout of traces on a printed circuit board is essentially the same problem, as well.



Note that combinatorial complexity of this problem

$$(N-2)!$$

since ends points are fixed. This grows very fast with N.

Possible solutions

- Brute force - combinatorial
  - Try every possible combination of cities and choose the best one
  - Will work for the modest route with $N \leq 10$ or may be slightly more

Permutation generating algorithm

The below method goes back to 14th century India. It generates permutations in the lexicographical order [1].

1. find the largest index k such that $p(k) < p(k+1)$.
   - if no such index exists, the permutation is the last permutation.
2. find the largest index l such that $p(k) < p(l)$.
   - There is at least one $l = k + 1$
3. swap $a(k)$ with $a(l)$.
4. reverse the sequence from $a(k+1)$ up to and including the final element $a(end)$.

See the complimentary code `permutation.m`

[1] See "The Art of Computer Programming, Volume 4 : Generating All Tuples and Permutations" by Donald Knuth for the discussion of the algorithm