Combinatorial optimization

Eugeniy E. Mikhailov

The College of William & Mary

Lecture 17

Notes

Combinatorial optimization problem statement

We still want to optimize (minimize) our multidimensional 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 $\vec{x}$.

Notes

Combinatorial optimization problem statement

We still want to optimize (minimize) our multidimensional 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 $\vec{x}$. Well, actually we can but constructing a proper constraining function will be a nightmare.

Notes

Combinatorial optimization problem statement

We still want to optimize (minimize) our multidimensional 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 $\vec{x}$. Well, actually we can but constructing a proper constraining function will be a nightmare.

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

Hence, the name combinatorial optimization.

Notes

Example: Backpack problem

Suppose you have a backpack with the given size (volume) and a set of objects with given volumes and monetary (or sentimental) values.

Our job is to find a 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.
Our job is to maximize

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

Subject to the following constraints

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

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

Notes

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 then the number of all possible combinations is 2^N.

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

Notes

Backpack optimization: new test set generation

Recall that the $\vec{x}$ component is either 0 or 1.
So, $\vec{x}$ 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?

Notes

Backpack optimization: new test set generation

Recall that the $\vec{x}$ component is either 0 or 1.
So, $\vec{x}$ 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 the new packed value is larger than 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.

Notes

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];
[items_to_take, max_packed_value] = ...
  backpack_binary( backpack_size, volumes, values)

items_to_take = [1 3 4]
max_packed_value = 78
```

Notes

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];
[items_to_take, max_packed_value] = ...
  backpack_binary( backpack_size, volumes, values)

items_to_take = [1 3 4]
max_packed_value = 78
```

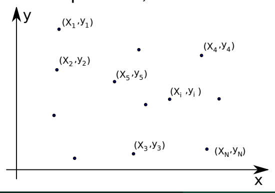
- My computer sorts through 20 items in 47 seconds, but 30 items would take 1000 times longer, i.e. about 13 hours to solve.

Notes

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 N_{th} city at the end of a route.
- The salesman must visit every city and do it only once.
- Find the shortest route which satisfies the above conditions.

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 a similar problem. The traces placement on a printed circuit board is essentially the same problem, as well.

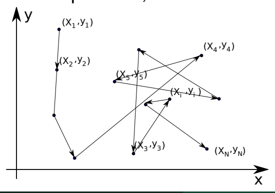


Notes

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 N_{th} city at the end of a route.
- The salesman must visit every city and do it only once.
- Find the shortest route which satisfies the above conditions.

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 a similar problem. The traces placement on a printed circuit board is essentially the same problem, as well.

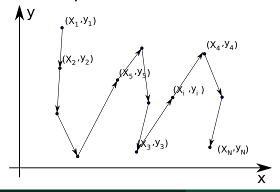


Notes

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 N_{th} city at the end of a route.
- The salesman must visit every city and do it only once.
- Find the shortest route which satisfies the above conditions.

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 a similar problem. The traces placement on a printed circuit board is essentially the same problem, as well.

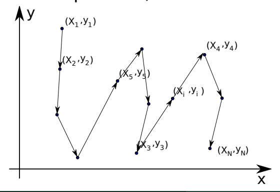


Notes

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 N_{th} city at the end of a route.
- The salesman must visit every city and do it only once.
- Find the shortest route which satisfies the above conditions.

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 a similar problem. The traces placement on a printed circuit board is essentially the same problem, as well.



Since the ends points are fixed, the combinatorial complexity of this problem

$$(N-2)!$$

This grows very fast with the N increase.

Notes

Possible brute force solution

- Try every possible combination (permutation) of the cities ordering and choose the best one.
- Will work for the modest cities number $N \leq 10$ or may be slightly more.

Notes

Permutation generating algorithm

The method below goes back to 14th century India[1].

1. Start with the set sorted in the ascending order, i.e. $p = [1, 2, 3, 4, \cdots, N-2, N-1, N]$
2. Find the largest index k such that $p(k) < p(k+1)$.
   - If no such index exists, the permutation is the last permutation.
3. Find the largest index l such that $p(k) < p(l)$.
   - There is at least one $l = k+1$
4. Swap $p(k)$ with $p(l)$.
5. Reverse the sequence from $p(k+1)$ up to and including the final element $p(end)$.
6. We have a new permutation. If we need another, repeat from the step 2.

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

Notes