Notes - Algorithms & Data Structures - Greedy Algorithms - Part 2

- Dr Nick Hayward

A brief intro to greedy algorithms and associated solutions.

Contents

Intro

A key consideration for working with algorithms is the identification of problems that have no *fast* algorithmic solution.

An awareness of such *NP-complete* problems is a particularly useful skill to develop, and certainly beneficial in algorithm design and development.

To help with such problems, we may often consider *approximation* algorithms. In effect, options we may use to quickly define an approximate solution to an *NP-complete* problem.

We may also consider *greedy* strategies, which provide simple options and patterns for the resolution of such problems.

Approximation algorithms

When we deal with such *NP-complete* problems, we commonly begin by considering *greedy* algorithms as a *good enough* solution.

Such *Greedy algorithms* will give us an approximated solution, which will often be a good, usable solution.

For example, if we consider the above *set-covering* problem, we may define a working *greedy* algorithm as follows

- select a station that covers the most states in the country
    - the set needs to cover states that have not already been covered
    - it's acceptable for the set to cover some states with existing coverage
- then, repeat this selection process until all the states are covered...

This is an example of an *approximation algorithm*. We know that a complete calculation to find the exact solution will simply take too long. Instead, an approximation algorithm will give us a working solution in a useful amount of time.

However, we may still compare and judge such *approximation algorithms*.

For example, we may commonly check the following

- their speed - i.e. how fast they are in calculating a workable solution...
- the quality of the approximation - i.e. how close is the result to the expected optimal solution

Greedy algorithms are a useful and beneficial choice for such problems because they are simple to design and quick to execute.

For example, for the *set-covering* problem we may see a performance time of `O(n^2)` where `n` defines the number of *base stations*.

code example

We may now consider a coded example for the above set-covering problem.

To help with this example, we may use a subset of the defined *states* and *base stations*.

The first thing we need to consider is a *list* for the states, which includes those needed for the service's coverage.

```
# set of states for checking base station coverage
# set used to ensure no duplicate entries
states = set(["az", "ca", "id", "mt", "nv", "or", "ut", "wa"])
```

We can use a `set` for this list of states to ensure we do not have duplicate entries for the data.

Likewise, we also need to store a list of the base stations we may select for coverage.

```
# define hash table for the stations
base_stations = {}
# add station with state coverage
base_stations["station_one"] = set(["or", "nv", "ca"])
base_stations["station_two"] = set(["wa", "id", "mt"])
base_stations["station_three"] = set(["ca", "az"])
base_stations["station_four"] = set(["id", "nv", "ut"])
base_stations["station_five"] = set(["nv", "ut"])
```

We'll use a hash table to help structure the states relative to each base station with the keys as the individual station names.

Again, we can use a `set` for the states per station.

Then, we need to define an empty `set` we may use to store the stations for the final coverage. These will be the suitable stations we've identified during the execution of the algorithm.

calculate the stations

So, we now need to perform the calculation to determine the base stations required for the network coverage. The least number of stations required for state coverage in the country.

As we're working with *approximation* algorithms, we will commonly see multiple possible solutons to this calculation.

The goal of the calculation is to determine the best station for the required state coverage.

So, we can update our current code as follows

```
# define current best base station
best_base_station = None
# all states per base_station not yet covered...
states_covered = set()
```

The `states_covered` variable is a `set` for the states that a given base station may cover, which have not yet been covered.

We may then use a standard `for` loop to check every base station to determine the best option for our network coverage.

For example,

```
# check each station in base stations hash table - find best option
for base_station, states_per_station in base_station.items():
    # create an intersection of sets...
    covered = states & states_per_station
    # check set intersection
    # - does this station cover more states than current best statio...
    if len(covered) > len(states_covered):
        # record best base station option
        best_base_station = base_station
        # update states now covered...
        states_covered = covered
```

n.b. for further details on *Sets*, please review the extra notes on *set* data structure and usage.

In the above code, we may see a *set intersection*,

```
# create an intersection of sets...
covered = states & states_per_station
```

In effect, we now have an updated *set* with states in both `states` and `states_per_station`. The variable `covered` now includes previously uncovered states, which are now covered by this base station.

We can then check this station against the current best base station to see if it covers more states.

```
# check set intersection
# - does this station cover more states than current best statio...
```

```
    if len(covered) > len(states_covered):
        # record best base station option
        best_base_station = base_station
        # update states now covered...
        states_covered = covered
```

If that check returns true, the current base station will now become the best station.

We let the loop iterate through, and then add `best_base_station` to the current final list of base stations.

```
final_base_stations.add(best_base_station)
```

After the current checks for base stations, we need to update the running check for `states_needed`. In effect, we remove the states we now have covered from the states that still need coverage.

```
states -= states_covered
```

This loop may continue, of course, until there are no states left that need coverage. i.e. `states_needed` is now empty.

So, the final code for loop is as follows

```
# while states still exist to check...
while states:
    # define current best base station
    best_base_station = None
    # all states per base_station not yet covered...
    states_covered = set()
    # check each station in base stations hash table - find best option
    for base_station, states_per_station in base_stations.items():
        # create an intersection of sets...
        covered = states & states_per_station
        # check set intersection
        if len(covered) > len(states_covered):
            # record best base station option
            best_base_station = base_station
            # update states now covered...
            states_covered = covered

    states -= states_covered
    final_stations.add(best_base_station)
```

If we execute this algorithm for the defined states and base_stations, we get the following selection of stations

```
{'station_three', 'station_two', 'station_one', 'station_four'}
```

performance of greedy algorithm

If we now check the run time of this greedy algorithm, we'll see how it compares favourably to a perceived exact algorithm

no. of base stations	exact algorithm - O(n!)	greedy algorithm - $O(n^2)$
5	3.2 seconds	2.5 seconds
10	102.4 seconds	10 seconds
100	$4x10^{21}$ years	16.67 minutes