Comp 460 - Algorithms & Complexity

Spring Semester 2020 - Week 14

Dr Nick Hayward

Final Assessment

Course total = 30%

- continue to develop your app concept and prototypes
  - *working app*
    - must implement algorithms and data structures
  - *explain design decisions*
    - describe patterns used in design and development of app
    - structures, organisation of code and logic
  - *explain testing and analysis*
  - *show and explain implemented differences from DEV week*
    - where and why did you update the app?
    - perceived benefits of the updates?
  - *how did you respond to peer review?*
- anything else useful for final assessment...
- consider outline of content from final report outline
- ...

All project code must be pushed to a repository on GitHub.

n.b. present your own work contributed to the project, and its development...

Final Report

Report due on Tuesday 28th April 2020 @ 6.45pm

- final report outline - coursework section of website
  - *PDF*
  - *group report*
  - ***extra individual report*** *- optional*
- include repository details for project code on GitHub

greedy algorithms - intro

- a key consideration for working with algorithms
  - *identification of problems that have no fast algorithmic solution*
- awareness of such *NP-complete* problems
  - *a particularly useful skill to develop*
  - *certainly beneficial in algorithm design and development*
- to help with such problems
  - *often consider approximation algorithms*
- i.e. options we may use to quickly define an approximate solution
  - *e.g. to an NP-complete problem*
- may also consider *greedy* strategies
  - *provide simple options and patterns for resolution of such problems*

Video - Algorithms and Data Structures

NP-complete problems - intro

Algorithms - NP-Complete Problems - intro - UP TO 36:02

Source - Algorithms - YouTube

Algorithms and Data Structures

greedy algorithms - sample problems

- to help us consider such problems
  - *review some common examples to help conceptualise such resolution patterns.*
- e.g. review the following well-known problems
  - *classroom scheduling problem*
  - *knapsack problem*
  - *set-covering problem*
  - *...*

classroom scheduling problem

- a classroom is available for lectures
- want to ensure we can schedule as many classes as possible
  - *schedule during a defined time period*
- i.e. interested in optimal use of resources
  - *within a finite, constrained period of time...*

classroom scheduling problem - worked example - part 1

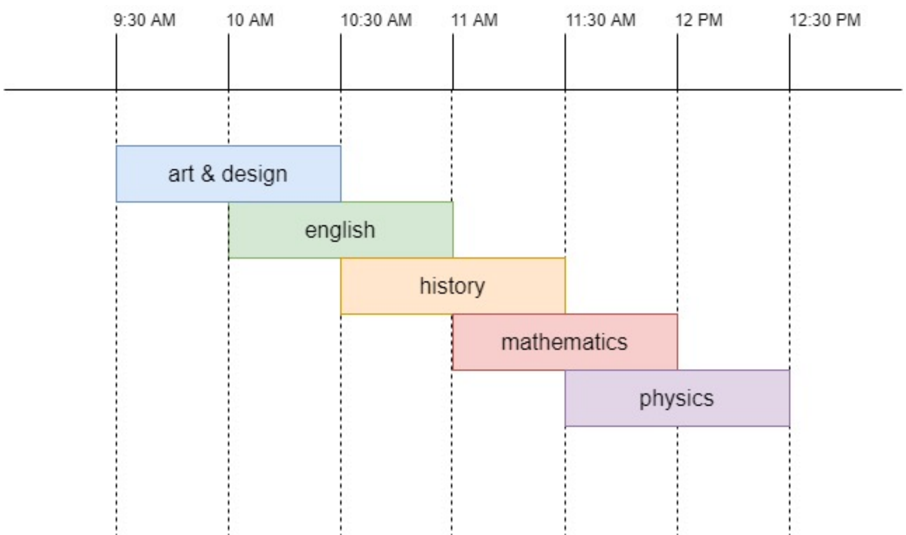
- begin by defining each class and its current scheduled hours
- e.g.

class	start time	end time
art & design	9:30 AM	10:30 AM
english	10 AM	11 AM
history	10:30 AM	11:30 AM
mathematics	11 AM	12 PM
physics	11:30 AM	12:30 PM

- as we can see in this table
  - *cannot currently schedule each of these classes in the classroom*
  - *there are time overlaps*
  - *and scheduling issues...*

classroom scheduling problem - worked example - part 2

- want to able to schedule as many classes as possible
  - *i.e. in this classroom*
  - *need to manage following schedule*
  - *ensure we fit most classes in current available time*
- e.g. current schedule is as follows
  - *including overlapping classes*



Classroom schedule

classroom scheduling problem - algorithm requirements - part 1

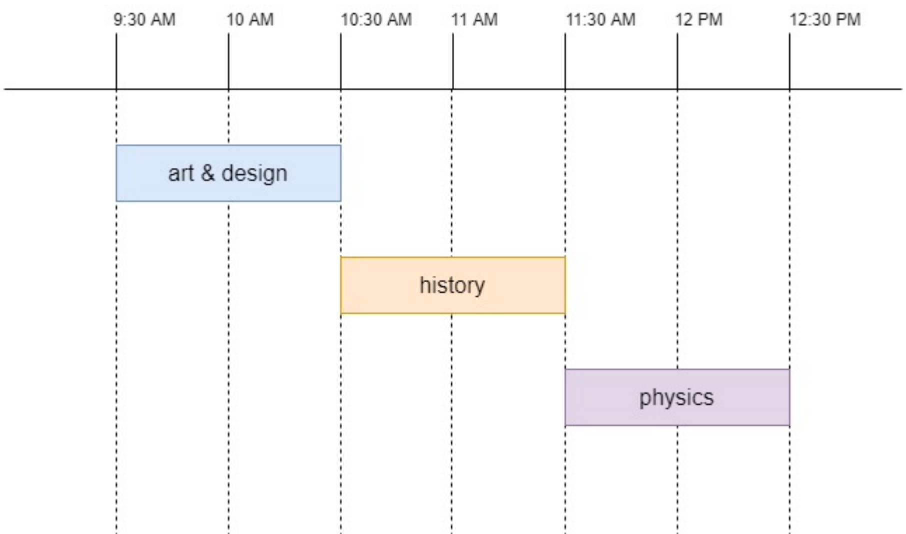
- define an algorithm to solve this problem for scheduling the classes
- whilst it may, initially, seem like a difficult problem to solve
  - *the algorithm is deceptively simple...*
- e.g. we may conceptually define this algorithm as follows
  - *select class that ends soonest...*
    - now the first class scheduled
  - *then, select a class that starts after this first class*
    - again, choose the class that ends soonest...
  - *repeat this pattern until schedule is full*
    - no more class will fit...

classroom scheduling problem - algorithm requirements - part 2

- if we apply this basic algorithmic solution
  - *update our classroom schedule as follows*
- *1. **art & design** - 9:30 AM to 10:30 AM*
  - *from our current classes, Art & Design finishes soonest*
  - *add that to our updated schedule*
- then, we need to identify a class that starts after 10:30 AM
  - *and, again, ends soonest of available classes*
- *2. **history** - 10:30 AM to 11:30 AM*
- repeat these checks
  - *update the schedule with the next class*
- *3. **physics** - 11:30 AM to 12:30 PM*

classroom scheduling problem - algorithm requirements - part 3

- now identified classes we may schedule for this classroom
  - *i.e. during the available timescale*



Classroom schedule

- whilst this algorithm may appear overly simplistic for a difficult problem
  - *we can see a clear benefit of greedy algorithms*
  - *they are easy to implement for such problems...*

classroom scheduling problem - algorithm requirements - part 4

- if we conceptualise a *greedy* algorithm
    - *at each step we're choosing the optimal selection*
- for this worked example
    - *simply picking a class*
    - *a class that ends soonest from matching options*

Algorithms and Data Structures

classroom scheduling problem - algorithm requirements - part 5

- as a developer, at each step of the algorithm
  - *choosing optimal local solution*
- this will then produce, at the end of the algorithm
  - *a globally optimal solution*
- this simple algorithm is now able to find optimal solution
  - *i.e. to this scheduling problem*
- *greedy* algorithms may not solve all problems
  - *but they are simple to write and test...*

knapsack problem

- another similiar example is the *knapsack problem*
- commonly perceived as an example of
  - *resource allocation*
  - *combinatorial optimisation*
- knapsack problem is conceptually simple to define and understand
  - *given a group of items - each with known value and weight*
  - *need to determine number of items we may fit in a given knapsack*
  - *knapsack of fixed size and capacity*
- i.e. need to calculate combined weight of these items
  - *ensure optimised collection is less than or equal to a set limit*
- likewise, need to ensure combined value is as high as possible…
- there are known constraints and requirements
  - *allow us to calculate optimal distribution of items*
  - *and associated best use of knapsack*

knapsack problem - worked example

- common example for this problem
  - *a burglar who needs to choose best goods*
  - *goods that will fit in their knapsack*
- burglar needs to grab a collection of items
  - *items with highest value*
  - *items they can carry in their bag*
- e.g. knapsack is able to carry a weight up to 20 kilograms
  - *approximately 44 pounds*
  - *trying to maximise total value of items carried in this bag*

knapsack problem - algorithm requirements - part 1

- if we consider an algorithmic solution
  - *might initially consider a greedy approach*
  - *use to try and solve this problem...*
- e.g.
  - *begin by picking item with highest value that will fit in bag*
  - *then, pick next expensive item that will fit in the bag*
  - *then repeat...*

Algorithms and Data Structures

knapsack problem - algorithm requirements - part 2

- **n.b.** this approach will not work for this example problem
  - *consider the following items*

item	weight	value
TV	15 kg	$2500
Computer	10 kg	$1500
Violin	7 kg	$1200

- we know the bag can carry up to 20 kg of items
- we can see most expensive item is the *TV*
  - *add that to the knapsack*
- it also weighs 15kg
  - *we may not add any of the other items.*

knapsack problem - algorithm requirements - part 3

- bag currently has a weight of 15kg with a value of $2500
- using this approach the highest value we may add is `$2500`
- clearly see that this is not best combination of items
- if we choose the *Computer* and *Violin*
  - *the value of the knapsack would now equal `$2700`...*

knapsack problem - algorithm requirements - part 4

- *greedy* strategy does not give an optimal solution to this problem
- if we consider the outcome
  - *it comes very close to the optimal solution*
- i.e. a quick use of this strategy will often be good enough to solve such problems
- for many problems
  - *an algorithm may solve the problem quickly and to a good enough standard*
- i.e. in this example
  - *only lost out on a potential* `$200`
  - *the calculation was fast and easy to execute*
- this type of scenario is where *greedy* algorithms prove very useful
  - *easy to write, and quick to execute...*

set-covering problem

- a related example for considering use of *greedy* algorithms
  - *commonly referred to as the set-covering problem*
- another *NP-complete* problem
  - *particularly useful as we consider approximation algorithms in general*
- outline of the problem is, again, deceptively simple to consider and understand
- e.g. a defined set of elements and a collection of sets
  - *these sets, when unified, same as initial set of elements*
  - *commonly known as the universe*
- problem requires identification of smallest union of sets
  - *union known to be equal to the universe...*

set-covering problem - worked example - part 1

- consider a problem to check for mobile internet coverage in a country
  - *coverage provided by a network of base stations in each state*
- internet coverage used to create a company
  - *company provides mobile data coverage for whole country*
- want to offer this service at lowest possible cost
  - *requires low setup and coverage costs*
- customer should be able to use service anywhere in country
- network service with full coverage across each of country's states
- trying to minimise number of *base stations*
  - *i.e. stations needed to be able to create a working, country-wide network...*

Algorithms and Data Structures

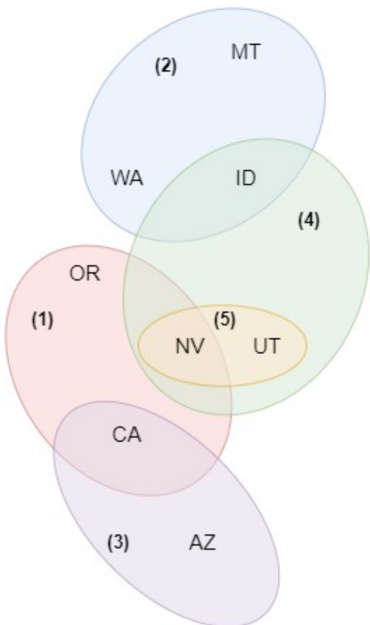
set-covering problem - worked example - part 2

- begin by compiling a sample of *base stations*
  - *those stations available to our company and network*
- e.g.

base station	state coverage
station one	OR, NV, CA
station two	WA, ID, MT
station three	CA, AZ
station four	ID, NV, UT
station five	NV, UT
...	,,,

set-covering problem - worked example - part 3

- clearly see that each station covers a given region of states
  - *also some overlap between stations and states*



Set Covering - Overlapping Base Stations

- need to calculate smallest set of *base stations*
  - *i.e. smallest set to cover required country area*
- may seem a simple problem to solve
  - *in practice, a difficult and time consuming problem to resolve...*

set-covering problem - algorithm requirements - part 1

- to solve this problem use following initial outline
- outline used to determine a set of *base stations*
- e.g.
  - *define each and every available subset of base stations for given coverage area*
    - commonly known as *power set*
    - 2^n possible subsets for this problem
  - *choose set with smallest number of base stations*
    - i.e. stations that meet coverage requirements for defined area
    - e.g. base stations for country

set-covering problem - algorithm requirements - part 2

- problem is not the calculation itself
  - *long time to calculate each and every potential matching subset of stations*
- it takes `O(2^n)` time
- dealing with `2^n` base stations
- calculation will be feasible for a smaller set of base stations
- this time quickly becomes impractical
- algorithm no longer a working solution to this problem....

Algorithms and Data Structures

set-covering problem - algorithm requirements - part 3

- e.g.

number of base stations	required calculation time
5	3.2 seconds
10	102.4 seconds
100	4×10^{21} years
...	....

- need to find a way to deal with such problems
- a solution that provides a working approximation
    - *and in a time useful for practical application...*

approximation algorithms - part 1

- when we deal with such *NP-complete* problems
  - *commonly begin by considering greedy algorithms*
  - *act as a good enough solution*
- *greedy algorithms* give us an approximated solution
  - *often a good, usable solution...*
- e.g. consider the *set-covering* problem
  - *define a working greedy algorithm*
- algorithm as follows
  - *select a station that covers most states in country*
    - set needs to cover states that have not already been covered
    - acceptable for set to cover some states with existing coverage
  - *then, repeat this selection process until all states are covered...*
- this is an example of an *approximation algorithm*

Algorithms and Data Structures

approximation algorithms - part 2

- know that a complete calculation to find exact solution takes too long
- approximation algorithm gives us a working solution
  - *and in a useful amount of time*
- may still compare and judge such *approximation algorithms*
- e.g. 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
  - *simple to design and quick to execute*
- e.g. for the *set-covering* problem
  - *may see a performance time of* `O(n^2)`
  - *where* `n` *defines number of base stations*

Video - Algorithms and Data Structures

approximation algorithms - heuristics & airports - part 1

Algorithms - Approximation & Heuristics - intro - UP TO 45:16

Source - Algorithms - YouTube

approximation algorithms - code example - part 1

- now consider a coded example for above *set-covering* problem
- to help with this example
  - *use a subset of defined states and base stations*
- first thing we need to consider is a *list* for states
  - *includes those needed for 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"])
```

- use a `set` for this list of states
  - *ensure we do not have duplicate entries for data...*

approximation algorithms - code example - part 2

- also need to store a list of base stations
  - *i.e. stations we may select for coverage*

```python
# 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"])
```

- use a hash table
  - *helps structure states relative to each base station*
  - *keys as individual station names*
- use a `set` for states per station

approximation algorithms - code example - part 3

- need to define an empty `set`
  - *use to store stations for final coverage*
- i.e. suitable stations identified during execution of algorithm

```
final_stations = set()
```

approximation algorithms - code example - part 4

- need to perform calculation to determine required base stations
  - *stations required for network coverage*
  - *least number of stations required for state coverage in the country*
- working with *approximation* algorithms
  - *commonly see multiple possible solutons to this calculation*
- goal of calculation is to determine best station for required state coverage
- update current code as follows

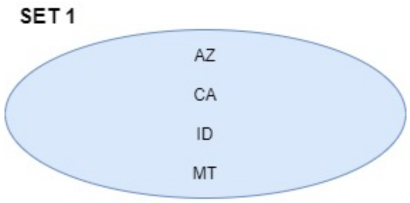
```
# define current best base station
best_base_station = None
# all states per base_station not yet covered...
states_covered = set()
```

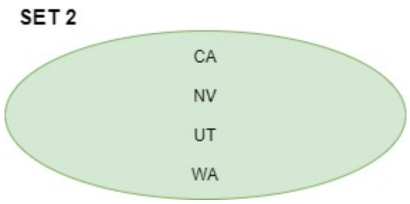
sets - intro

- a brief, but useful, segue into *Sets*
- `set` is an abstract data type
  - *stores unique values*
- no pre-defined, discernible order to the data stored
  - *n.b. data must be unique*
- data is a working implementation of a mathematical *finite set*
- unlike many other data structures
  - *do not customarily retrieve a specific element from a set*
- check *set* for existence of a given element
  - *unique record may then be used to retrieve required data*

sets - worked example - part 1

- might represent sets of items
  - *items will be unique to each set*
- may be duplication of elements in multiple sets
  - *but values must be unique per set...*
- e.g.

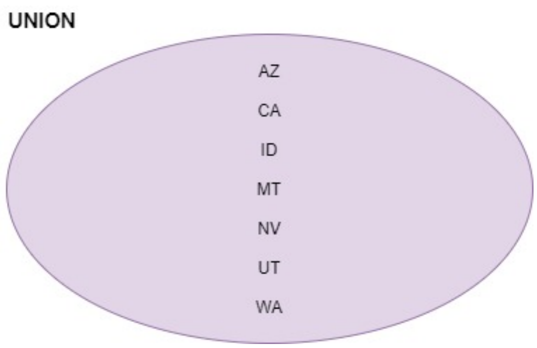




Two Sets of States

sets - worked example - part 2

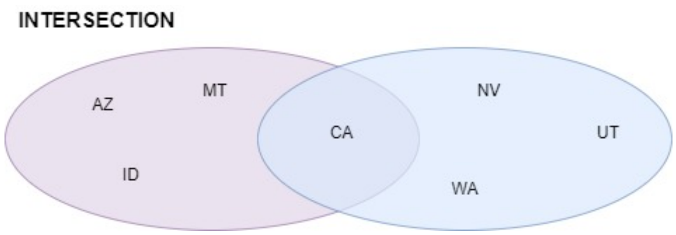
- then use such sets to perform various operations
- union
  - *a set containing all unique elements from a group of sets*
  - *combine sets to create a single unified set*
  - *e.g. union of set 1 and set 2*



Union of sets

sets - worked example - part 3

- intersection
  - *elements that exist in each of intersected sets*
  - *find elements that exist into all of defined sets*
  - *e.g. states that are in set 1 and set 2*

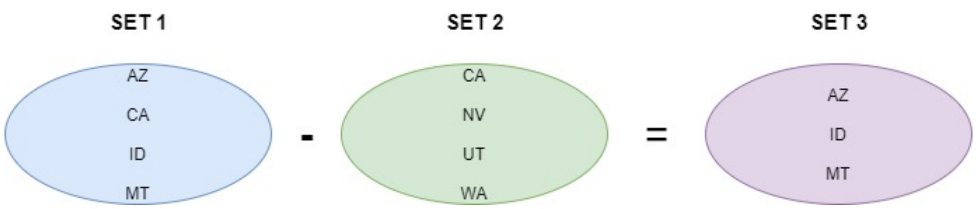


Intersection of sets

sets - worked example - part 4

- difference
  - *calculate difference between defined sets*
  - *subtract elements in one set from elements in another set*
  - *e.g. subract elements in set 1 from elements in set 2*

DIFFERENCE



Difference of sets

sets - code example - part 1

- implement such operations in code
- e.g. in Python we may use a set as follows

```python
states_set1 = set(["az", "ca", "id", "mt"])
states_set2 = set(["ca", "nv", "ut", "wa"])

# set union
states_union = states_set1 | states_set2
# set intersection
states_intersect = states_set1 & states_set2
# set difference
states_diff = states_set1 - states_set2
```

sets - code example - part 2

- then check results of these operations
  - *may see following output*
  - *e.g.*
- union of sets

```
{'ut', 'nv', 'az', 'mt', 'id', 'wa', 'ca'}
```

- intersection of sets

```
{'ca'}
```

- difference of sets

```
{'az', 'id', 'mt'}
```

Video - Algorithms and Data Structures

approximation algorithms - heuristics & airports - part 2

Algorithms - Approximation & Heuristics - Flight Management - UP TO 47:10

Source - Algorithms - YouTube

approximation algorithms - code example - part 5

- `states_covered` variable
  - *a `set` for states that a given base station may cover*
  - *i.e. thise not yet covered*
- then use a standard `for` loop
  - *check every base station to determine best option for network coverage*
- e.g.

```python
# 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
```

approximation algorithms - code example - part 6

- in example code, we may see a *set intersection*

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

- i.e. now have an updated *set*
  - *states in both `states` and `states_per_station`*
- variable `covered` now includes previously uncovered states
  - *i.e. now covered by this base station*

approximation algorithms - code example - part 7

- then check this station against current best base station
  - *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
  - *current base station will now become best station*

Algorithms and Data Structures

approximation algorithms - code example - part 8

- loop iterates through
  - *then add `best_base_station` to current final list of base stations*

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

- after current checks for base stations
  - *need to update running check for `states_needed`*
- i.e. remove states now covered from states that still need coverage

```
states -= states_covered
```

- loop may continue until there are no states left that need coverage
  - *i.e. `states_needed` is now empty...*

approximation algorithms - code example - part 9

- final code for loop is as follows

```python
# 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)
```

approximation algorithms - code example - part 10

- if we execute this algorithm for defined states and base_stations
    - *we get the following selection of stations*

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

performance of greedy algorithm

- check run time of this greedy algorithm
  - *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

np-complete - intro

- in *set-covering* problem
  - *need to calculate each possible set*
  - *regardless of the number of sets*
- common feature of *NP-complete* problems
  - *lack of a fast, exact algorithmic solution*
  - *i.e. as scale of problem increases*
- classic example for *NP-complete* problems is *Traveling Salesman* problem

np-complete - traveling salesman

- a salesman needs to visit a series of cities
  - *e.g. initially starting out from Cairo*
- salesman would like to visit these cities using shortest practical route
- to be able to calculate shortest route
  - *need to initially calculate each and every possible route*
- consider a trip that needs to visit five cities
  - *how many routes do we actually need to calculate?*

Video - Algorithms and Data Structures

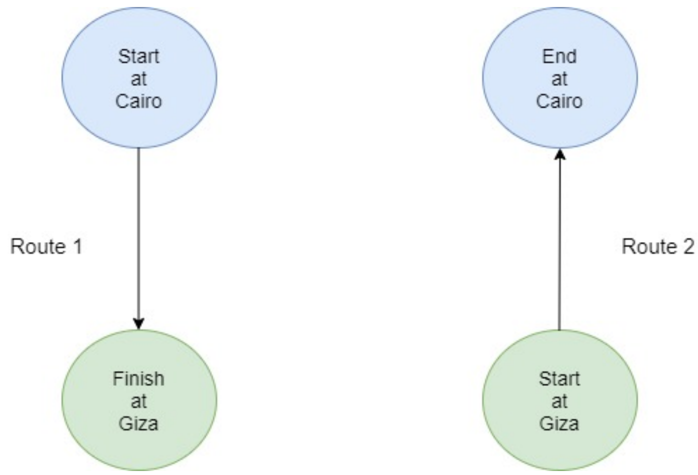
NP-complete - Traveling Salesman

Algorithms - NP-Complete - Traveling Salesman - UP TO 38:40

Source - Algorithms - YouTube

np-complete - traveling salesman - two cities - part 1

- begin with a simple calculation
  - *initially only two cities in the trip*
  - *quickly calculate two possible routes salesman may choose for this trip*



Traveling Salesman - 2 cities

- consider these routes
  - *might initially question why there is a duplication*
  - *aren't these routes the same?*

np-complete - traveling salesman - two cities - part 2

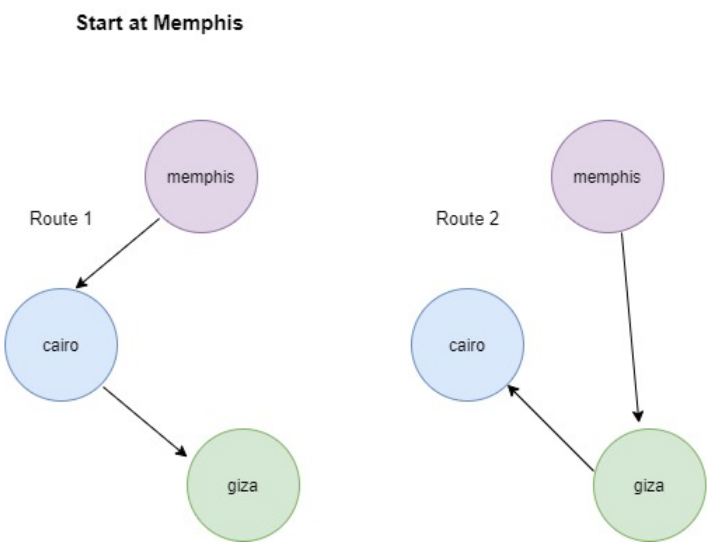
- inherent problem
  - *cannot be certain each route is same distance, time, path, &c.*
  - *many routes will have one-way streets*
    - perhaps only heading north
  - *routes may have diversions due to planning requirements...*
  - *different highways will also have different access ramps depending upon direction of travel*
  - *...*
- i.e. need to be recorded as two separate routes
- other common query
  - *whether we need to ensure we begin at a given city in network...*

np-complete - traveling salesman - two cities - part 3

- current example begins in Cairo
  - *cannot assume this will always be true for each salesman, every trip...*
- salesman may need to begin in Cairo, Giza, Memphis &c.
- may be a delay in travel
  - *need to restart at a different city &c.*
- an assumption we cannot hold as true
- start location is unknown
  - *algorithm needs to be able to compute optimal path for salesman*
  - *optimal path regardless of origin*

np-complete - traveling salesman - three cities - part 1

- then add a third city to current trip
  - *need to revise calculation to consider number of possible routes*
- e.g. start at *Memphis*
  - *two cities to visit*
  - *including Cairo and Giza*



Traveling Salesman - 3 cities

np-complete - traveling salesman - three cities - part 2

- with a starting point at Memphis
  - *two possible routes to Cairo and Giza*
- similar pattern may be seen if we begin at either Cairo or Giza
  - *returning two possible routes for each starting position*
- for *three* cities we have **six** possible routes

np-complete - traveling salesman - four cities

- add a fourth city to trip
  - *may continue calculation for possible routes*
- may add *Saqqara* as a city the salesman needs to visit during this trip
- start trip at this new city, *Saqqara*
  - ***six** possible routes*
- quickly see a pattern emerging
  - *defines six available routes per available starting point*
- with four possible start cities
  - *six possible routes for each start*
  - *a simple calculation of* `4 x 6 = 24` *possible routes*
- each time we add a new city
  - *increasing number of routes we need to calculate for trip*

np-complete - traveling salesman - add more cities - part 1

- add more cities
  - *start to see how possible number of routes will grow rapidly*
- e.g.

no. of cities	possible routes
1	1 route
2	2 start cities x 1 route for each start = 2 total routes
3	3 start cities x 2 routes = 6 total routes
4	4 start cities x 6 routes = 24 total routes
5	5 start cities x 24 routes = 120 total routes
6	6 start cities x 120 routes = 720 total routes
7	7 start cities x 720 routes = 5040 total routes
8	8 start cities x 5040 routes = 40320 total routes
...	...

np-complete - traveling salesman - add more cities - part 2

- a clear pattern to growth of possible routes relative to defined number of start cities
- known as **factorial function**
  - *e.g.* `5! = 120`
- check total number of possible routes for 10 cities
  - *calculate a total as* `10!`
  - *equals* `3,628,800`
- for just 10 cities in a route
  - *need to calculate over three million possible routes*
- number of possible routes become very large, very quickly as calculation executes
- currently not feasible to compute a *correct* solution for this problem
  - *i.e. if there is a high number of cities in trip*

Video - Algorithms and Data Structures

algorithms - ongoing use and application

Algorithms - Ongoing use and application - UP TO END

Source - Algorithms - YouTube

Resources

various

- Approximation algorithms - Wikipedia
- How the Mathematical Conundrum Called the ‘Knapsack Problem’ Is All Around Us - Smithsonian Magazine
- Knapsack problem - Wikipedia
- Networking - Set-covering problem - MIT
- NP-complete - Wikipedia
- NP-complete - NIST
- Python - Sets
  - *Sets - Python.org*
  - *Sets - W3Schools*
- Set-covering problem - Wikipedia
- Traveling Salesman - Wikipedia

videos

- Heuristics and Airports
  - *part 1 - intro - up to 45:16*
  - *part 2 - heuristic algorithm - up to 47:10*
- NP-Complete problems - intro - up to 36:02
- Ongoing use and application - up to end
- Traveling Salesman Problem - up to 38:40