Data Structures & Algorithms 2

Topic 8 – Weighted Graphs

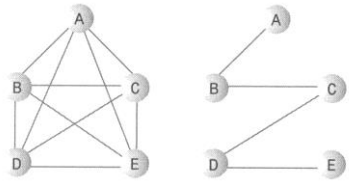
Weighted Graphs

- Now to introduce an additional edge feature which makes things more complicated: weight
- For example, if vertices in a weighted graph represent cities then the weight of the edges might represent the distances between the cities or the cost to fly between them
- Interesting question arise such as
  - What it the minimum spanning tree (MST) for a weighted graph?
  - What is the shortest (or cheapest) distance between two vertices?
- These questions have very important applications in the real world

Minimum Spanning Tree

- Suppose you want to connect all the vertices in your graph using the least number of edges
- This is called a *minimum spanning tree* (MST)
- The fewest number of edges needed will always happen to be one less than the number of vertices
  - E = V – 1
- For unweighted graphs this is simple –just use a searching algorithm to visit every vertex
- For weighted graphs it is a bit more complex

Minimum Spanning Tree



a) Extra Edges

b) Minimum Number of Edges

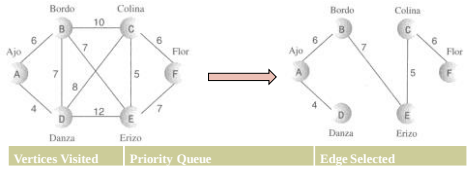
Minimum Spanning Tree

- Because the edges have different weights, it is important to choose the right selection in order to link all of the vertices
- A real world application of the MST algorithm would be to link a series of towns up to the electricity grid
- What is the cheapest way to link them all together?

The MST Algorithm (Prim's algorithm)

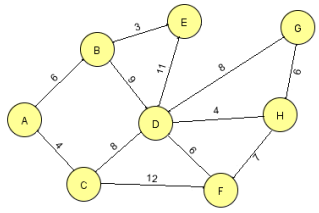
1. Start with any vertex and add it to the minimum spanning tree
2. Add newly available edges to a priority queue
3. Pick the edge with the lowest weight, remove it from the priority queue, and print it out as part of the solution. The vertex it leads to is added to the minimum spanning tree
4. You don't want to visit a vertex more than once. Therefore, when you visit a vertex make sure to remove all of the remaining edges that connect already-visited vertices
5. Keep going until every vertex has been visited. The graph is now completely connected

Example



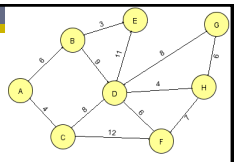
Vertices Visited	Priority Queue	Edge Selected
A	AD4, AB6	AD4
A D	AB6, DB7, DC8, DE12	AB6
A D B	BE7, DC8, BC10, DE12	BE7
A D B E	EC5, EF7, DC8, BC10	EC5
A D B E C	CF6, EF7	CF6
A D B E C F	-	-

Example



Find a minimum spanning tree
Start at A

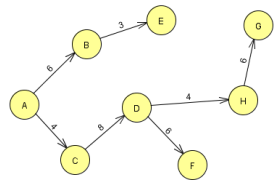
Worked Solution



Vertices Visited	Priority Queue	Edge Selected
A	AC4, AB6	AC4
A C	AB6, CD8, CF12	AB6
A C B	BE3, CD8, BD9, CF12	BE3
A C B E	CD8, ED11, CF12	CD8
A C B E D	DH4, DF6, DG8, CF12	DH4
A C B E D H	DF6, HG6, HF7, DG8, CF12	DF6
A C B E D H F	HG6, DG8	HG6

Just keep selecting the shortest edge in the priority queue
Then add in the new potential edges leading from that vertex

Solution



7 edges needed to connect 8 vertices
All vertices connected using minimum weight

Dijkstra's algorithm

- Dijkstra's algorithm is also known as the shortest path problem
- The problem was conceived by Dutch computer scientist Edsger Dijkstra in 1959
- This algorithm is often using for routing network traffic in the most efficient manner
- The problem is also applicable to wide variety of real-world situations from the layout of printed circuit boards to project scheduling

Dijkstra's algorithm

- The essentials of Dijkstra's algorithm are as follows:

1. Start at any vertex and consider the weights on the edges leading from it
2. Always visit the unvisited vertex that has the total cheapest path from the starting point
3. Each time you visit a new vertex, use the new edge information to revise your list of cheapest paths to each unvisited vertex from the starting point
4. Keep going until all vertices have been visited

Implementation

- We need an array that keeps track of the minimum distances from the starting vertex to the other vertices
- During execution these distances are changed until at the end they hold the actual shortest distances for all vertices from the start vertex
- We create an array that holds the shortest known distances to all of the vertices involved
- This Shortest-Path Array simply has one slot for each of the vertices in the graph

A	B	C	D	E
-	50	100	80	140

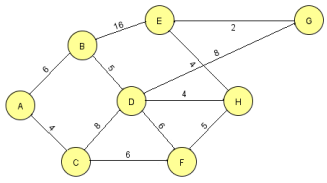
Implementation

- As well as tracking the minimum distance to each vertex, we also need to know the path taken to get there
- Luckily, we don't have to store the whole path – just the previous step
- For example, if we know the shortest path to E came through C then we just have to look where the shortest path to C came through etc.
- Use the Shortest-Path Array to store the shortest distance to each vertex and the previous step on the shortest path to that vertex

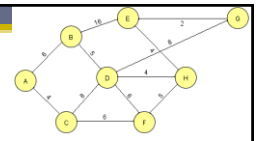
A	B	C	D	E
-	50(A)	100(D)	80(A)	140(C)

Example

- Use Dijkstra's shortest path algorithm to find the cheapest path between the vertices A and E



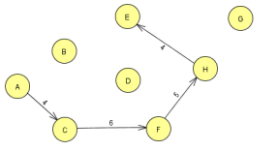
Worked Solution



Vertices Visited	A	B	C	D	E	F	G	H
A	0	6(A)	4(A)	-	-	-	-	-
A C	0	6(A)	4(A)	12(C)	-	10(C)	-	-
A C B	0	6(A)	4(A)	11(B)	22(B)	10(C)	-	-
A C B F	0	6(A)	4(A)	11(B)	22(B)	10(C)	-	15(F)
A C B F D	0	6(A)	4(A)	11(B)	22(B)	10(C)	19(D)	15(F)
A C B F D H	0	6(A)	4(A)	11(B)	19(H)	10(C)	19(D)	15(F)
A C B F D H E	0	6(A)	4(A)	11(B)	19(H)	10(C)	19(D)	15(F)

- Keep picking the shortest path to an unvisited vertex (red shows visited)
- When you move to a new vertex update the shortest paths

Solution



- The final row in the Shortest-Path Array represents the shortest paths to all of the vertices when starting at A

Vertices Visited	B	C	D	E	F	G	H
A C B F D H E G	6 (A)	4 (A)	11 (B)	19 (H)	10 (C)	19 (D)	15 (F)

All-pairs Shortest Path

- Dijkstra's algorithm gives us the shortest distance from one vertex to all the other vertices in the graph
- We can easily enhance this algorithm so instead of just a single dimension array it produces a matrix which gives the shortest distance from any vertex to any other vertex
- All we do is run Dijkstra's algorithm several times, starting from a different vertex each time
- This gives us the all-pairs shortest-path matrix

All-pairs Shortest Path

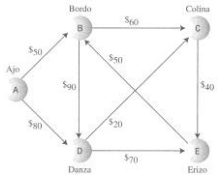


TABLE 14.11 All-Pairs Shortest-Path Table

	A	B	C	D	E
A	—	50	100	80	140
B	—	—	60	90	100
C	—	90	—	180	40
D	—	110	20	—	60
E	—	50	110	140	—

Efficiency

- Efficiency depends on whether we're representing the graph using an adjacency matrix or an adjacency list
- If we use an adjacency matrix the algorithms mostly require $O(V^2)$ time, where V is the number of vertices
- The reason for this is because there is a slot in the matrix for a potential edge between a vertex and every other vertex
- We have to check through all of these potential edges whether there is a valid edge or not
- If there are many vertices this is bad news – however if the graph is dense (meaning the graph has many edges) then most of the slots in the adjacency matrix are filled anyway and there's not much we can do to improve performance

Efficiency

- Many graphs are *sparse* meaning that there are few edges and most of the slots in an adjacency matrix would be empty
- In this case running time can be improved by using an adjacency list
- Adjacency lists just hold a list of all the existing edges so you don't waste any time checking matrix slots that don't hold edges

TABLE 13.1 Adjacency Matrix

	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

TABLE 13.2 Adjacency Lists

Vertex	List Containing Adjacent Vertices
A	B—>C—>D
B	A—>D
C	A
D	A—>B

Efficiency

- For un-weighted graphs the depth-first search with adjacency lists requires O(V+E) time where V is the number of vertices and E is the number of edges
- For weighted graphs both the minimum spanning tree and shortest-path algorithms require O((E+V)logV) time using an adjacency list

Intractable Problems

- We have seen algorithms so far with varying complexity, from O(1) up to $O(n^2)$
- These algorithms can easily be run for n values into the millions
- However, some algorithms have big O values that are much greater (for example, 2^n)
- Many real world problems are of this nature and simply cannot be solved in a reasonable amount of time - these problems are said to be *intractable*

P-problems (easy to solve)

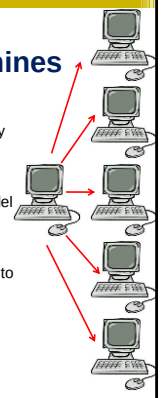
- Problems are assigned to classes depending on the complexity of the algorithm required to solve them
- A problem is assigned to the P (polynomial time) class if there exists at least one algorithm to solve that problem with a complexity of $O(n^x)$ where x is some number
  - $n^2 + 3n$
  - $n^5 + n^3 + \log n$
  - $n^{10000} + 16$
- These are considered 'easy' problems to solve

EXP-problems (difficult)

- A problem is assigned to the EXP (exponential time) class if the fastest algorithm to solve that problem has a complexity of $O(2^{p(n)})$ where p(n) is a polynomial expression of n (e.g. 2^{6n})
- An EXP problem is like trying to guess a password where there are no short-cuts except to try them all
- Example: finding out if a Turing machine halts in at most *n* steps: we must try every possible set of *n* steps
- EXP means brute force is the only way to go – we can use an imaginary non-deterministic machine to do this

Non-deterministic machines

- A non-deterministic Turing machine is an imaginary 'magical' machine that is able to evaluate multiple possibilities at the same time
- The machine branches into multiple different parallel worlds and the solution is returned as soon as it is found by one of the branches
- A non-deterministic Turing Machine would be able to crack a password by evaluating every possible permutation simultaneously in different worlds
- It tries every possibility at the same time

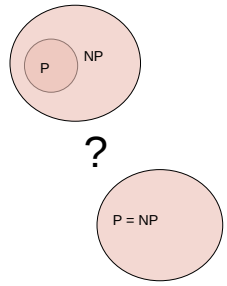


NP-problems

- A problem is assigned to the complexity class NP if a solution can be computed by a non-deterministic polynomial machine and the solution can be recognized in P time by an ordinary Turing machine
- The $1,000,000 question: are there really any problems which can be quickly recognized in P time but which cannot be solved in P time? (that is, the solution can only be computed by brute force using a non-deterministic polynomial machine)
- In other words, are there some problems in NP (answers can be easily recognized) that are not in P? (cannot be easily solved)

NP-problems

- The question P ?= NP is one of the most important questions left to be answered in mathematics
- Are there really some problems whose solutions are easier to verify than they are to compute?
- Currently there is no evidence either way except that some NP problems seem really difficult yet have easily verifiable solutions



If P = NP...

- If you could find a P solution for any NP-complete problem it would change the fields of mathematics and computer science forever
- Every password and code in existence could be cracked in a feasible amount of time – internet security would be impossible
- Every mathematical theorem that has a finite proof could be proved in a feasible amount of time – all questions with answers could be answered
- Some mathematicians believe that P != NP must be a fundamental property of the universe - if P was equal to NP it would undermine the intuitive concepts of space and time

P versus NP

- Examples of things which seem difficult to do yet easy to appreciate:
  - Writing a great song
  - Writing a great book
  - Proving a mathematical theorem
  - Cracking a password

- If these things were easy to do then nothing would be valuable
  - A great book could hold no value because it would be just as easy to write it as to read it
  - No goal would be 'far away'
  - Life would be meaningless!

NP-problems

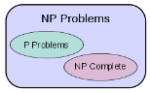
- There are many mathematical problems that have easy-to-verify solutions (i.e. are in NP) but *seem* to be difficult to solve (i.e. are not in P)
- For example, find the factors of 7,279,690,869,631
  - It is thought that solving the factorisation problem is $O(k^{\log n})$
  - But maybe there is a quick way to do it?
  - A solution is 472,517 x 2,944,243
  - The solution is easy to verify
- Can we be sure that the solution cannot be found quickly? Can we prove it?

Clay Institute Millennium Prize

- The Clay Mathematics Institute is offering a €1 million reward for answering the P =? NP question one way or the other as one of the 8 Millennium Prize problems
- Of course, if P = NP you could use your algorithm to find proofs for the other 7 Millennium prize problems, therefore collecting €7 million instead of €1 million!

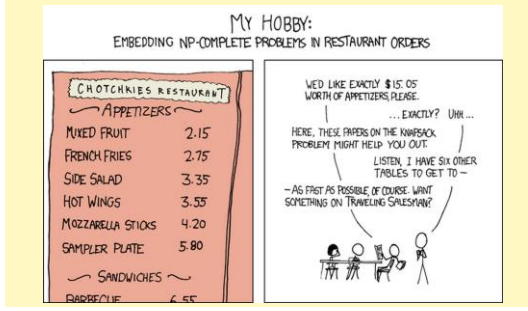
NP-complete problems

- NP-complete problems are the hardest possible NP problems
- If any NP-complete problem can ever be solved by a polynomial algorithm, then P must equal NP because it would mean that even the hardest NP problems would be easy to solve
- All NP-complete problems are really the same problem – they can be reduced to each other in polynomial time
- Nobody has ever found a polynomial time algorithm to solve any of them



Clicker Question

A)	YES
B)	NO

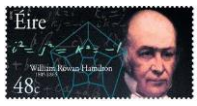


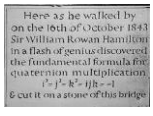
NP-complete example (subset-sum)

- The subset-sum problem has been proven to be NP-complete
- Does any subset of these numbers sum to 53?
  - (15, 8, 14, 26, 32, 16, 9, 22)
- Solutions:
  - 15 + 22 + 16
  - 22 + 14 + 9 + 8
  - These solutions are hard to find yet easy to verify!

William Rowan Hamilton (1805 – 1867)

- Hamilton was an Irish physicist, astronomer and mathematician
- He made contributions to classical mechanics, optics and algebra
- In 1843 he suddenly came up with the idea of quaternions suddenly while out for a stroll along the Royal canal
- He carved his idea into the stone of Broom Bridge in Cabra





Traveling Salesman Problem

- The Traveling Salesman Problem (TSP) is another famous NP-complete problem that was defined by Hamilton
- You're given a list of cities on a map and you want to visit each one covering the least distance possible
- Unfortunately, the list of possible routes is very large – it's the factorial of the number of towns
  - 6! = 6 x 5 x 4 x 3 x 2 x 1 = 720
- The problem becomes impractical to solve for even 40 cities

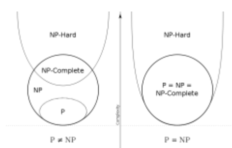
Traveling Salesman Problem

The TSP problem is the minimum spanning tree for a weighted graph problem where every vertex is linked to every other vertex and the weights are determined by the positions of the vertices on a 2-dimensional map. In the solution, each vertex must have exactly two edges leading from it

- You need to find the shortest path that links all of the vertices but there are n! number of edges to consider
- If you can find an algorithm for solving this problem whose complexity is bounded by a polynomial (e.g. n^2, n^3, n^4 etc.) then you can then arrange to pick up your cheque for $1,000,000
- Otherwise (unless you've got a few million years to spare) you're going to need to design a fast algorithm which finds a good (though not optimal) solution

TSP decision is NP-complete

- Say somebody makes a claim – *"there is a TSP path for this problem that is under 1600km"*
- It seems difficult to verify if this is correct or not
- However, if they show you the path, it is easy to verify if they are telling the truth
- This is why the TSP decision problem is NP-Complete
- The TSP optimisation problem is NP-Hard, which means it is at least as hard to solve as the hardest NP problems, but may not be in NP (solution is as hard as possible to compute but not necessarily easy to recognize)
- If any NP-Hard problem can be solved in P time then so can all NP and thus P=NP
- For $1,000,000, can you find a quick solution for finding paths?



Your objective

- 80 towns and cities scattered around Ireland have been selected
- Your objective is to find a way to visit each town while covering the least amount of distance and arrive back at your starting point

TSP records

- 1954 – 49 city problem solved
- 1977 – 120 city problem solved
- 1987 – 2,392 city problem solved
- 1992 – 3,038 city problem solved
- 1998 – 13,509 city problem solved
- 2004 – 24,978 city problem solved
- 2006 – 85,900 city problem solved (current record holder)

Distance Matrix

- You have been given the GPS co-ordinates of the towns in terms of latitude and longitude
- Based on this you have computed a distance matrix which gives the distances between all of the towns

	Wexford	Waterford	Sligo	Shannon Airport	Rosslare	Roscommon	Portlaoise	Limerick	Killarney	Kilkenny	Galway	Dundalk	Dublin	Donegal	Derry	Cork	Belfast	Athlone
Belfast																		141
Cork																	264	136
Derry																266	73	130
Donegal															43	250	112	114
Dublin														138	147	160	104	78
Dundalk													53	98	97	202	52	90
Galway												148	136	127	169	130	190	58
Kilkenny											107	123	73	192	208	92	177	78
Killarney										123	120	219	192	253	274	54	271	144
Limerick									69	70	65	150	123	184	204	65	201	75
Portlaoise								71	140	32	93	94	52	160	175	108	157	46
Roscommon							66	94	164	98	51	94	91	94	131	156	139	20
Rosslare						150	84	131	171	62	170	153	101	243	247	129	205	130
Shannon Airport					146	96	86	15	84	85	57	165	138	176	218	80	215	83
Sligo				136	203	53	119	144	213	152	86	104	135	41	84	209	128	73
Waterford			182	95	51	129	62	80	120	30	137	151	98	222	238	78	207	108
Wexford		39	191	133	12	138	71	118	157	50	157	141	88	231	235	116	192	117
Wicklow	56	88	162	156	72	123	77	141	208	77	168	85	32	171	179	116	136	110

Distance Chart in Miles

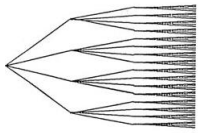
Worst Strategy

- Just pick random towns that have yet to be picked
- You end up with a random sequence of towns
- You might have found a good path but it will most likely be awful
- You could loop the program and store the best solution found so far

Brute Force Search

- You could simply run a naïve brute-force search
- There are 80! possible paths to be evaluated
  - That's 7.15 x 1^{115} possibilities
  - The universe is only 4.32 x 10^{17} seconds old
- You wouldn't use breadth-first search because that would only find a solution after considering every single possibility
- Depth-first search would search all the way down taking one particular path at each point until it finds one solution and then it would backtrack and start finding other solutions
- Although depth-first will quickly come up with a solution, the universe would expire before the best solution was found

Branching Factor



- The above tree has a branching factor of 4 (4 options at each level)
- Chess has a branching factor of around 35
- The project has a branching factor of 80!
- We need some heuristics to speed up the search

Greedy algorithms

- We could use a greedy algorithm like nearest neighbour
- Keep choosing to go to the nearest town first
- Just use your matrix and mark out towns you've visited so they are not visited again (you can do this by setting all the edges leading to that town to a huge number, like a million kilometres, so they never get selected)
- You'll probably find that you finish up quite a distance from Maynooth at the second last step

A clever random idea

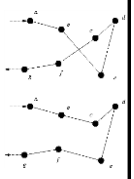
- Let the computer do the work rather than your brain
- Combine the greedy search with a random element
- For example, add in a random number from 0 – 20km to all the slots in your matrix
- Now a slightly different answer will be generated each time
- Just keep track of the best solution found so far and let your algorithm run overnight!

A better greedy algorithm

- Another alternative is to rank all of the distances between all towns
- Keep adding in the shortest distance between any two unvisited towns on the map
- The one condition is that you want to doubling back on yourself
  - Don't include links to towns you have included already
  - For example, you don't want to end up back in Maynooth before completing a full tour

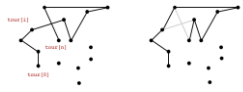
2-opt

- 2-opt is a good algorithm for enhancing a given solution even further
- Undoing crossovers always shortens the overall distance. No optimal solution can have a crossover
- The idea is to take points in the path that cross over – then uncross them
- Simply take any two edges in your solution and try swapping them – if the path has shortened then you have eliminated a crossing
- Repeat for all possible edge pairings and all crossing are eliminated



2-opt

- Getting rid of a crossover is equivalent to reversing a chunk of your path
- For example 45.24.67.26.15.75 might turn into 45.**15.26.67.24**.75 by reversing the four in the middle.
- To get rid of a crossover just try reversing every possible internal chunk of your path and keep the solution that is shortest. The crossover should be gone.
- Why not combine a clever greedy algorithm with added Monte Carlo random jitter, polished off by 2-opt?



Held-Karp

- Proposed in 1962 by Bellman and independently by Held and Karp
- It is based on dynamic programming
- Dynamic programming is a method for solving a complex problem by breaking it down into a collection of simpler subproblems
- Dynamic programming avoids resolving the same subproblems over and over again by storing the solution in memory in lookup tables
- It is an example of a space-time tradeoff

Held-Karp

- Every subpath of a path of minimum distance is itself of minimum distance
- Compute the solutions of all subproblems starting with the smallest
- Whenever computation of a solution requires solutions for smaller problems, look up these solutions which are already computed
- The time complexity is $O(2^n n^2)$ and the space is $(2^n n)$

Held-Karp

Held-Karp

- You encode sub-solutions in terms of where you are now, and which towns have left to be visited

- $g(1,\{2,3,4\}) = \min \{ c_{12} + g(2,\{3,4\}), c_{13} + g(3,\{2,4\}), c_{14} + g(4,\{2,3\}) \}$
- $= \min \{2 + 20, 9 + 12, 10 + 20\}$
- $= \min \{22, 21, 30\}$
- $= 21$

Linear Programming

- TSP can also be formulated as an integer linear program
- Label the cities with numbers 0 to n and define

$$x_{ij} = \begin{cases} 1 & \text{the path goes from city } i \text{ to city } j \\ 0 & \text{otherwise} \end{cases}$$

- Now we just need to solve the matrix, with the constraints that
  - 1) each city be arrived at from exactly one other city
  - 2) from each city there is a departure to exactly one other city
  - 3) the tour isn't broken up into separate 'islands'

Linear Programming

- You need to import software for solving linear equations
- CPLEX made by IBM has a Java interface and is very fast
- See this video for using Java and CPLEX https://www.youtube.com/watch?v=sf59_7r8QSY

$$\min \sum_{i=0}^{n} \sum_{j \neq i, j=0}^{n} c_{ij} x_{ij}$$

$$0 \leq x_{ij} \leq 1 \qquad i, j = 0, \cdots, n$$

$$u_i \in \mathbf{Z} \qquad i = 0, \cdots, n$$

$$\sum_{i=0, i \neq j}^{n} x_{ij} = 1 \qquad j = 0, \cdots, n$$

$$\sum_{j=0, j \neq i}^{n} x_{ij} = 1 \qquad i = 0, \cdots, n$$

$$u_i - u_j + n x_{ij} \leq n - 1 \quad 1 \leq i \neq j \leq n$$

DFS with Branch-and-Bound

- Branch-and-bound is a heuristic we can use to speed up depth-first search by not bothering to consider paths that couldn't possibly end up being better than the best solution found so far
- If a partial tour is already longer than the best solution found so far that there is no need to continue searching down that path

Upper Bound

- Run your depth-first search which quickly comes up with a solution
- Keep track of the best solution found so far and the distance involved (e.g. 2,140 km)
- If you algorithm is examining a partial solution which already exceeds 2,140km then there is no point in continuing searching that path
- Backtrack and pop the next partial solution off the stack

Better Branch-and-Bound

- We can make branch-and-bound even better by figuring out the lower bound for a partial path – the minimum size that any full solution is going to be based on a partial path
- We can then dump any partial paths for which the lower bound already exceeds the upper bound
- We don't even have to wait for a partial path to exceed the upper bound before dumping it if we know that it will definitely exceed the upper bound by the time the path is complete
- Obtaining the lower bound is done by examining the distance matrix and transforming it in some clever ways

Obtaining a Lower Bound

- We use the distance matrix for the towns remaining to be visited
- We are able to subtract a constant from any row or column without affecting the validity of the solution
- The outputted weights will change but the resulting solution will still be correct

ïj	1	2	3	4	5	6	7
1	Inf	3	93	13	33	9	57
2	4	Inf	77	42	21	16	34
3	45	17	Inf	36	16	28	25
4	39	90	80	Inf	56	7	91
5	28	46	88	33	Inf	25	57
6	3	88	18	46	92	Inf	7
7	44	26	33	27	84	39	Inf

Obtaining a Lower Bound

- We normalize the matrix by subtracting the minimum value in each row from the row and the minimum value of each column from the column
- This results in a matrix with at least one zero in every row and column

ïj	1	2	3	4	5	6	7
1	Inf	0	83	9	30	6	50
2	0	Inf	66	37	17	12	26
3	29	1	Inf	19	0	12	5
4	32	83	66	Inf	49	0	80
5	3	21	56	7	Inf	0	28
6	0	85	8	42	89	Inf	0
7	18	0	0	0	58	13	Inf

Obtaining a Lower Bound

- Any solution must use one entry from every row and every column, so the sum of the values we just subtracted is a lower bound on the size of the eventual complete path
- In this case we subtracted [3 4 16 7 25 3 26] from the rows and then [0 0 7 1 0 0 4] from the columns
- Without needing to evaluate the remainder of the path, we know there is no way the path could possibly be less than 96
- If this already exceeds the upper bound, then we can avoid considering any more possibilities involving this partial path

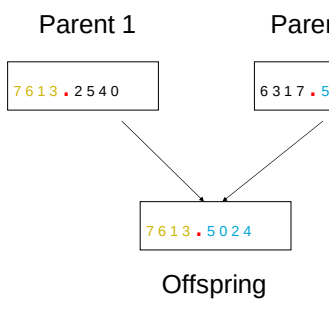
Obtaining a Lower Bound

- What have we actually done here?
- We've found the shortest distance taken to reach each city from any other city
- We've assumed that every city will be reached from the city nearest to it and figured out the cumulative distance in this case
- In reality, any actual solution will involve a longer distance than this
- As you build up your partial path, the rows and columns corresponding to towns you have already visited will become redundant
- This means that the lower bound will be constantly changing – it will increase as the path gets longer

Genetic Algorithms

- Genetic algorithms follow the idea of evolution
- Genetic algorithms have 4 parts
  - Generate (a random population)
  - Select (the fittest)
  - Crossover (breed two fit solutions to produce children)
  - Mutate (introduce some random variance)
- Because only the fittest (best) solutions get to breed, the population becomes very fit after only a few generations
- A good solution is produced very quickly but often the solutions become too "inbred" and get stuck on a sub-optimal configuration

Genetic Algorithms



A "GeneRepair" method is run to fix up the offspring solution if it leaves out a number or has the same number more than once