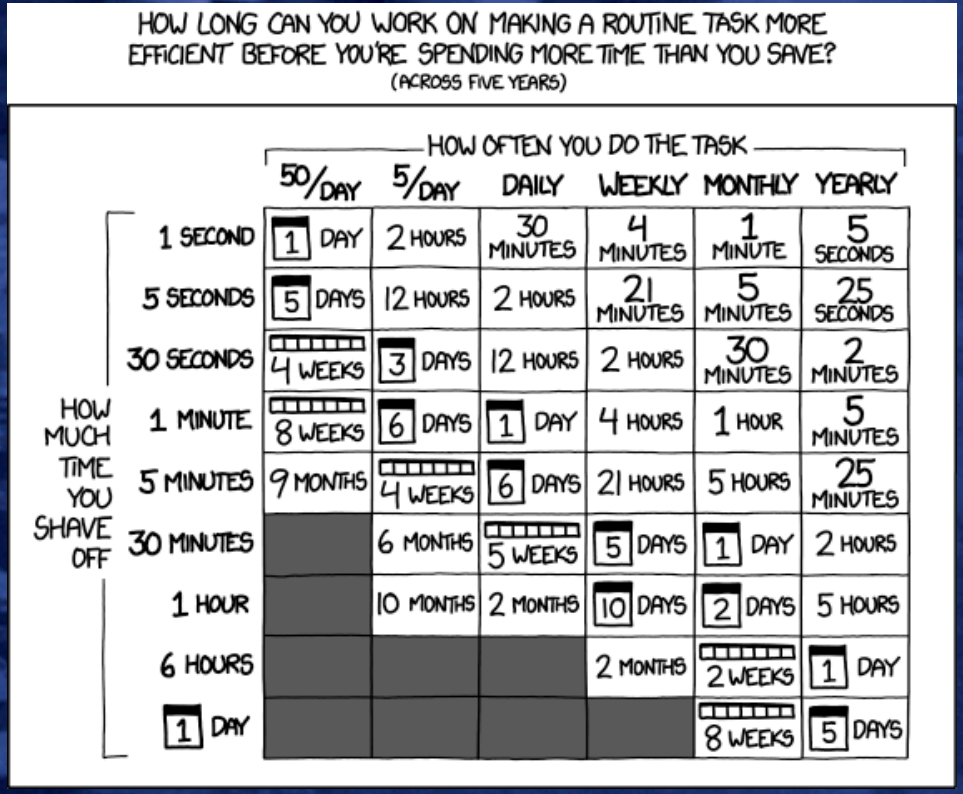
Optimisation

CITS3001 Algorithms, Agents and Artificial Intelligence



Tim French
Department of Computer Science and Software Engineering
The University of Western Australia

CLRS Chapters 34-35

2021, Semester 2

Summary

THE UNIVERSITY OF WESTERN AUSTRALIA

- We will define optimisation problems and we will look at four important algorithmic techniques for tackling these problems
  - Greedy algorithms
  - Dynamic algorithms
  - Approximation algorithms
  - Gradient-based search algorithms

- Many problems that we face are absolute
  - there is a right answer, and there are a lot of wrong answers
- e.g. when we are given a list of *n* distinct items to sort, there are *n!* permutations of the items, only one of which is sorted
  - The other *n!–1* permutations are simply wrong
- Sometimes (e.g. with longest common subsequence), there is a set of (equally) right answers, and a lot of wrong answers
- In both cases, the distinction between “right” and “wrong” is absolute

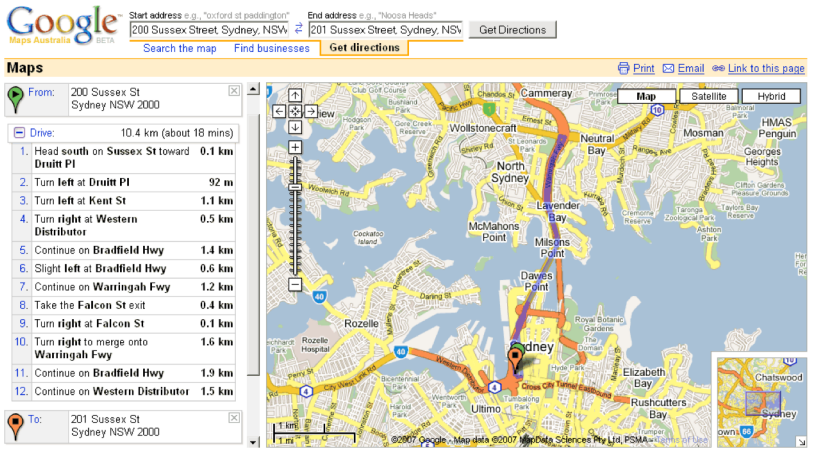
Optimisation Problems

THE UNIVERSITY OF WESTERN AUSTRALIA

- But a lot of problems are more complicated
- e.g. consider a piece of machinery whose speed of operation is controlled by a dial that can be set to any real number between 1 and 1,000
- How do we set the dial?
  - We could set it to 1,000 to go as fast as possible
    - But then the machine might wear out quickly
  - We could set it to 1 to extend the machine's life
    - But then it makes product very slowly
  - We could try to find some compromise setting in the middle of the range
    - Probably the usual approach
- The point for now is that no solution is "right", and no solution is "wrong"
  - Whatever we set the dial to, the machine runs
  - Just some settings work "better", and others work "worse"
  - The distinction is a matter of degree
- These problems are commonly called *optimisation problems*
  - We are trying to find a solution that performs well wrt some criterion (or criteria)

Example

- Imagine you have to find a path from Point *A* to Point *B* in a complicated road network
- There will probably be many different paths that will get you from *A* to *B*
  - So all of them are “solutions”
  - But one (or a few) of them will get you there quickest, or safest, or using the least fuel, …
  - i.e. some of them will be “better” wrt whatever criteria are important
- Note that many of these problems are often expressed in two different ways:
  - Find the best path
  - Find a path, and make it as good as possible
- The former phrasing obviously has a right answer in the absolute sense
  - The second does not
  - Hence only the second is an optimisation problem



Optimisation Algorithms

THE UNIVERSITY OF WESTERN AUSTRALIA

- We discuss four approaches to optimisation problems
- *Greedy algorithms* build up a solution bit-by-bit
  - At each step they make the locally-optimal choice for the next "bit"
- *Dynamic programming* we have seen before
  - Define recursive rules for solving the problem, then optimise the algorithm by eliminating repeated work
- *Approximation algorithms* focus on returning good solutions, but not necessarily the best one
  - Often they can operate very quickly
- *Gradient-based search algorithms* take known solutions and try to improve them
  - While they can be slow, they can work well in difficult problem domains

- Often there will be a trade-off between the run-time of the algorithm and the quality of the solution returned
  - We will see this issue again later in CITS3001
  - It is also an important issue with so-called "computational intelligence" approaches, studied in CITS4404

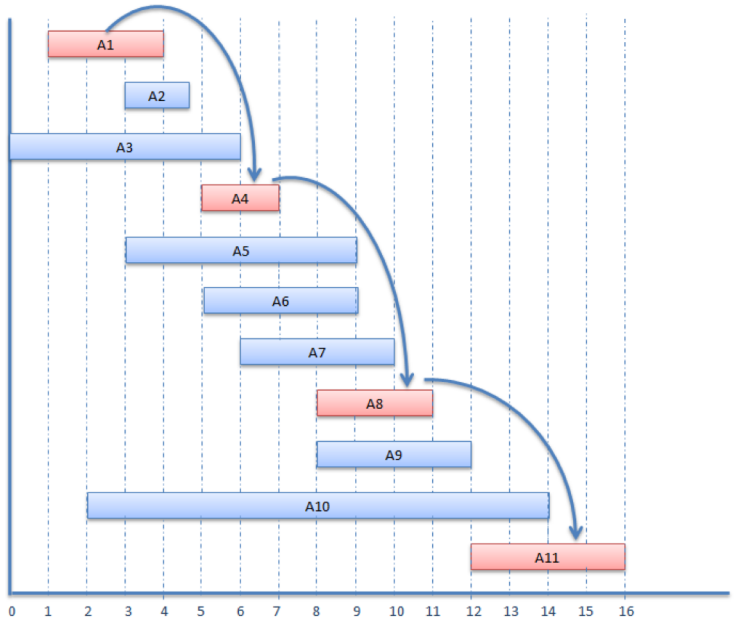
An activity selection problem

- Given a set of tasks, each with an associated start time and finish time, select the largest subset of the tasks that can be performed without any *incompatibilities*
  - Two tasks are incompatible if they overlap in time
- e.g. for {(6,9), (1,10), (2,4), (1,7), (5,6), (8,11), (9,11)}, the following schedules are all valid
  - {(1,10)}, {(1,7), (8,11)}, {(2,4), (5,6), (9,11)}
- We will assume that the activity intervals are *closed* on the left and *open* on the right
  - A closed end of an interval includes its endpoint
  - $(a,b) = \{x \in \mathbf{R} \mid a < x < b\}$
  - $[a,b) = \{x \in \mathbf{R} \mid a \leq x < b\}$
  - $(a,b] = \{x \in \mathbf{R} \mid a < x \leq b\}$
  - $[a,b] = \{x \in \mathbf{R} \mid a \leq x \leq b\}$
- The natural approach to the activity-selection problem is to choose the tasks one at a time
  - Clearly each choice restricts subsequent choices
- e.g. given {[6,9),[1,10),[2,4),[1,7),[5,6),[8,11),[9,11)}
  - If we (arbitrarily) choose [1,7), subsequent choices are restricted to {[8,11), [9,11)}
- This is just a smaller instance of the same problem
  - Effectively the activity-selection problem can be reduced to the question of (repeatedly) choosing one task from a set
  - Which suggests a greedy approach: try to make the locally-optimal choice at each stage

Greedy rules for activity-selection

THE UNIVERSITY OF WESTERN AUSTRALIA

- What greedy rules might be reasonable?
- Any of the following rules might be a candidate
  - Choose the shortest task
  - Choose the task that overlaps with fewest others
  - Choose the task that starts earliest
  - Choose the task that finishes earliest
  - Choose the task that starts latest
  - Choose the task that finishes latest
  - Or other possibilities, some much more complicated
- The greedy approach is a very local procedure
  - Make the choice that seems best now, without worrying about how it affects future choices
- Sometimes the greedy approach works well – sometimes it even produces optimal answers – but frequently it doesn't
- Which of these rules work? Can you prove or disprove that they work?



An optimal choice

THE UNIVERSITY OF WESTERN AUSTRALIA

- In this case, it turns out there is a provably-optimal greedy rule
  - Always choose the task that finishes earliest

- e.g. given {[6,9),[1,10),[2,4),[1,7),[5,6),[8,11),[9,11)}
  - Choose [2,4), leaving {[6,9),[5,6),[8,11),[9,11)}
  - Choose [5,6), leaving {[6,9),[8,11),[9,11)}
  - Choose [6,9), leaving {[9,11)}
  - Choose [9,11)

- The formal proof of optimality is by contradiction (CLRS Ps. 373–5), but intuitively:
  - Suppose there exists a solution $\{t_1, t_2, \ldots, t_k\}$ that does not include [2,4)
  - Assume that $t_1, \ldots, t_k$ are ordered by finish time
  - Clearly t_1 does not intersect with any of $t_2 \ldots t_k$
  - Clearly [2,4) finishes no later than t_1
  - Therefore [2,4) does not intersect with any of $t_2 \ldots t_k$

Running time for activity-selection

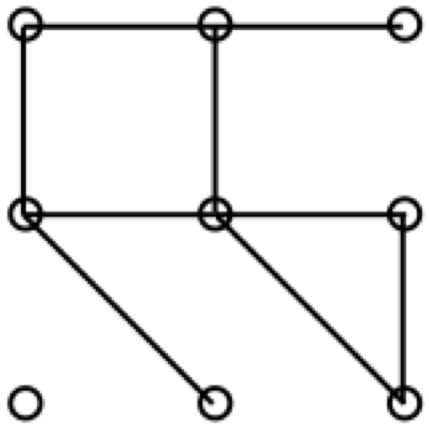
THE UNIVERSITY OF WESTERN AUSTRALIA

- The running time for this algorithm is dominated by the time to sort the tasks initially, i.e. O(*nlogn*)
- Greedy algorithms are often very fast, because they are usually very simple
- So why don't we always use the greedy approach?
  - Because it doesn't always work! (obviously…)
- There is a theorem identifying precisely the set of problems for which greedy algorithms work: They are known as *Matroids* (see CLRS 16.4), and include problems such as shortest path, minimum spanning tree and activity selection.
- Unfortunately there are many problems for which greedy algorithms (provably) do not work.

The Vertex Cover Problem

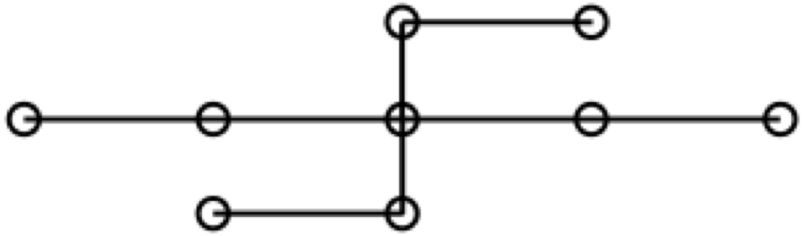
THE UNIVERSITY OF WESTERN AUSTRALIA

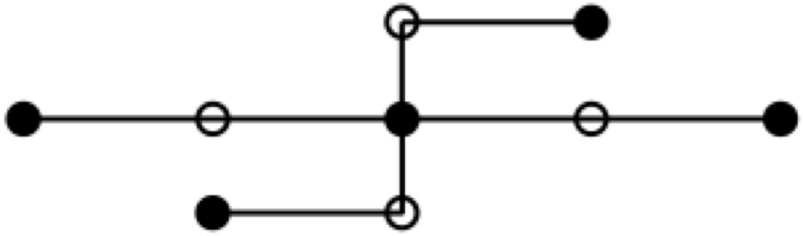
- A *vertex cover* for a graph *G* is a set of vertices $V' \subseteq V(G)$ such that every edge in *G* has at least one end in *V'*
  - We say that *V' covers* all the edges in *G*
- The Vertex Cover problem is to find the smallest vertex cover of *G*
- Exercise: What is the smallest vertex cover of the graph below?

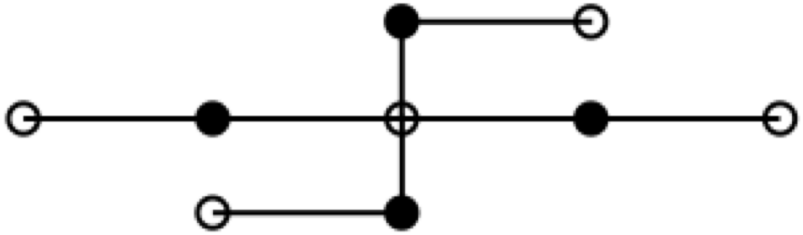


A Greedy Algorithm?

- One obvious greedy rule would be
  - At each stage, choose the vertex that covers the most remaining uncovered edges
- But for the graph below, this greedy rule would choose the middle vertex, whereas there is a better solution:



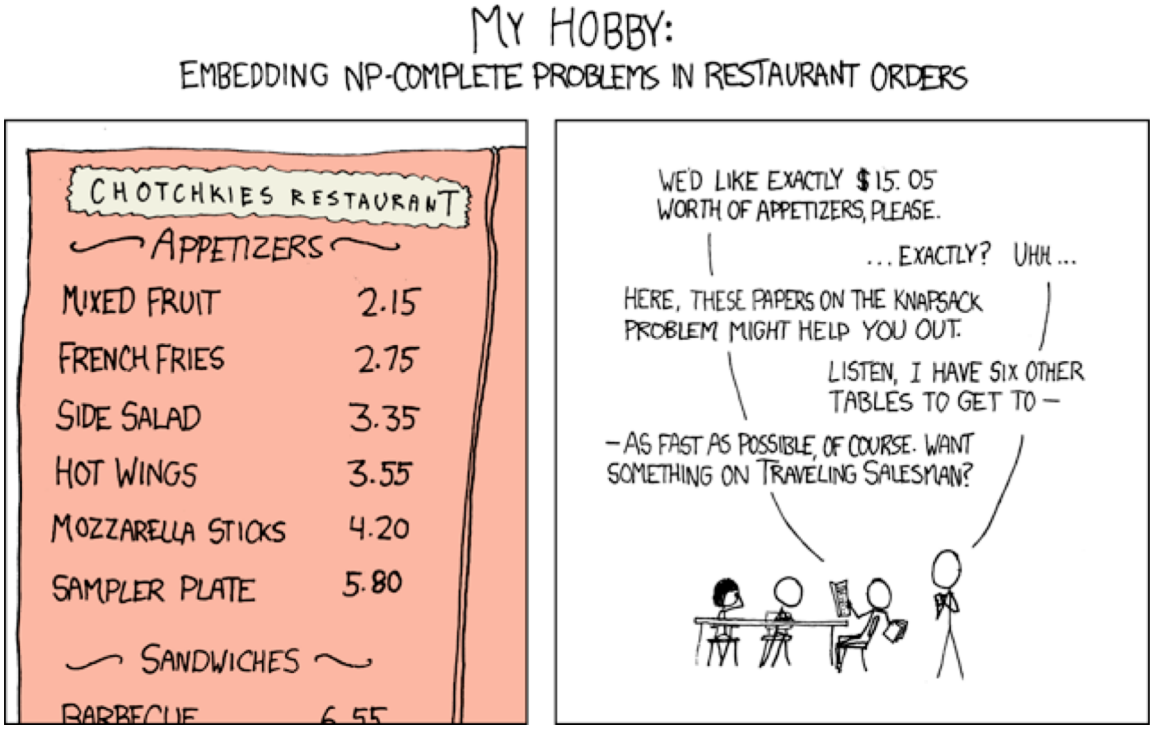




The problem with greedy algorithms

THE UNIVERSITY OF WESTERN AUSTRALIA

- The problem is that the locally-optimal choice (covering as many uncovered edges as possible) reduces our options for later choices
- Unfortunately most problems are *not* amenable to the greedy approach
- There is no known algorithm for Vertex Cover which is essentially better than just enumerating all possible subsets of the vertices
- Vertex Cover is an example of an *NP-hard* problem



P vs NP

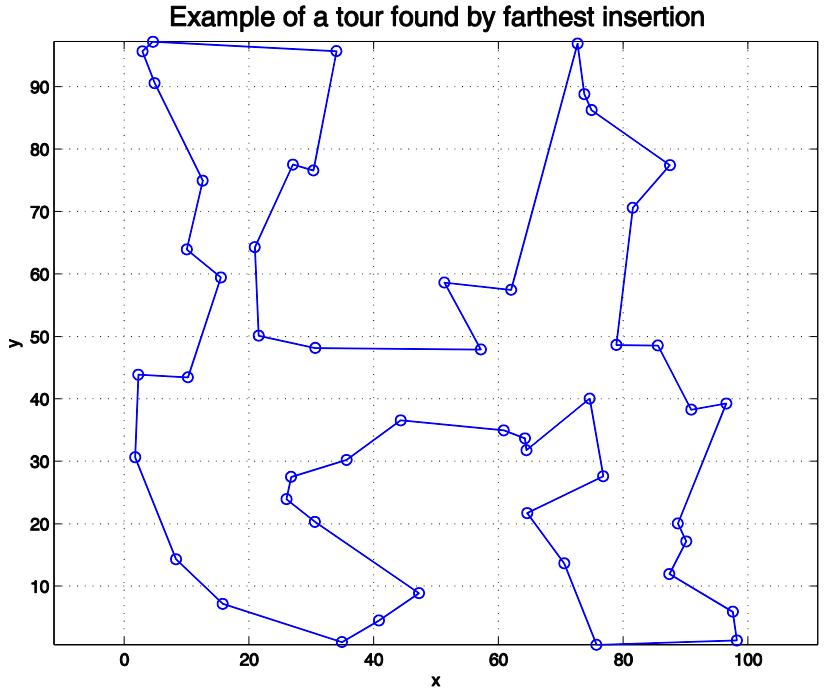
THE UNIVERSITY OF WESTERN AUSTRALIA

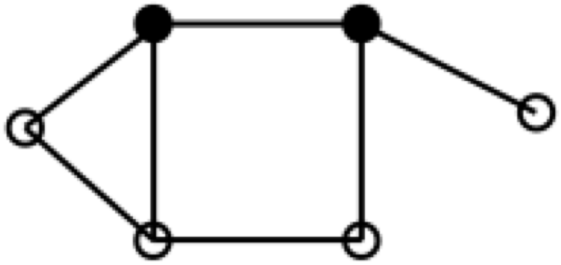
- A computational problem *x* is in the class **P** if there is a deterministic algorithm that solves *x* and that runs in polynomial time (i.e. $O(n^k)$ for some *k*)
  - These are polynomial-time problems, often called *feasible* or *tractable*
  - …even though for $k > 20$ they aren't really either

- A computational problem *x* is in the class **NP** if there is a non-deterministic algorithm that solves *x* and that runs in polynomial time
  - These are non-deterministic polynomial-time problems, often called *infeasible* or *intractable*
  - But these algorithms require lucky guesses to work efficiently
  - NP complete problems are a class of NP problems which are in NP and polynomially equivalent to one another. NP hard problems are problems at least as hard as NP complete problems.
  - Whether there is a deterministic polynomial algorithm to solve NP-complete problems is a well-known open problem.
  - In this unit , you will receive a disproportionate penalty if you ever refer to NP as "not polynomial".

Two more NP problems

Travelling Salesman (*TSP*): given a finite set of cities *C* and a distance function $d(c_i, c_j) \in \mathbf{R}^+$, find the shortest circular tour that visits each city exactly once .





Dominating Set: given a graph *G*, what is the smallest set of vertices $V' \subseteq V(G)$ such that every vertex in *V(G)* is adjacent to at least one vertex in *V'*

How hard are these problems?

THE UNIVERSITY OF WESTERN AUSTRALIA

- Every problem *x* in the NP-hard class has the following properties
  - There is no known polynomial-time algorithm for *x*
  - The only known algorithms take exponential time
  - If you could solve *x* in polynomial-time, then you could solve them all in polynomial-time
- Vertex Cover, Travelling Salesman, and Dominating Set are all NP-hard

- The most important open problem in theoretical computer science is whether or not this class of problems can be solved in polynomial-time
- Many more interesting complexity classes:

  https://complexityzoo.uwaterloo.ca/Complexity_Zoo

Longest common subsequence

THE UNIVERSITY OF WESTERN AUSTRALIA

Finally, we'll consider the problem of comparing string to see how similar they are…

Given two sequences X and Y, what is their longest common subsequence?

A subsequence of X is X with zero or more items omitted

e.g. ABC has seven subsequences: ABC, AB, AC, BC, A, B, C

A sequence with *n* items has $2^n - 1$ subsequences

So e.g. the LCSs of ABCBDAB and BDCABA are

BCBA

BCAB

BDAB

We can solve this problem efficiently using *dynamic programming*

A recursive relationship

The first step is to find a recursive rule whereby the main problem can be solved by solving smaller sub-problems

Assume that

$X = x_1\ x_2\ \ldots\ x_m$

$Y = y_1\ y_2\ \ldots\ y_n$

and that they have an LCS

$Z = z_1\ z_2\ \ldots\ z_k$

Clearly either $x_m = y_n$, or not

If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}

e.g. X = abcd, Y = paqd

If $x_m \neq y_n$, then at least one of x_m and y_n was discarded, and Z_k is an LCS of either X_{m-1} and Y, or X and Y_{n-1}

For this case there are three possibilities

e.g. X = abcd, Y = padq
(q discarded)

e.g. X = abcd, Y = apqc
(d discarded)

e.g. X = abcd, Y = apbr
(both d and r discarded)

The recursive relationship

This allows us to define a recursive relationship:

$$LCS(X_i, Y_j) = [\,], \quad \text{if } ij = 0$$
$$= LCS(X_{i-1}, Y_{j-1}) + [x_i], \quad \text{if } x_i = y_j$$
$$= longer(LCS(X_{i-1}, Y_j), LCS(X_i, Y_{j-1})), \quad \text{if } x_i \neq y_j$$

And a corresponding relationship on lengths:

$$len(X_i, Y_j) = 0, \quad \text{if } ij = 0$$
$$= len(X_{i-1}, Y_{j-1}) + 1, \quad \text{if } x_i = y_j$$
$$= max(len(X_{i-1}, Y_j), len(X_i, Y_{j-1})), \quad \text{if } x_i \neq y_j$$

Applying this rule directly as an algorithm requires the calculation of $LCS(X_i, Y_j)$ for all $0 \leq i \leq m$ and $0 \leq j \leq n$

It should also be clear that it requires the repeated calculation of some sub-expressions

e.g. LCS(abcd, pqr)

= longer(LCS(abc, pqr), LCS(abcd, pq))

= longer(longer(LCS(ab, pqr), LCS(abc, pq)),
longer(LCS(abc, pq), LCS(abcd, p)))

=

Dynamic Programming

THE UNIVERSITY OF WESTERN AUSTRALIA

These are the two essential features which tell us that dynamic programming can be applied

- Recursive sub-structure
- Overlapping sub-problems

The basic principle is known as *memoisation* and is very simple:

- When we evaluate an application, remember the result so that we don't have to evaluate it again

We maintain a table of applications that have been evaluated. For every new application, we check first to see if we have done it previously

- If we have, just look up the result
- If we haven't, do the evaluation and store the result

Dynamic programming applies this principle in a slightly cleverer way

- We know what applications have to be evaluated, so we order them in such a way that repetition is avoided and no checking is needed

Recall: Fibaonacci numbers

The recursive program

```
fib(k) = fib(k-1) + fib(k-2), if k > 1
= 1, if k <= 1
```

Recursive sub-structure and
overlapping sub-problems

The “dynamic programming” version

```
fib(k)       = fib'(k, 1, 1)
fib'(k, x, y)   = x, if k == 0
   = fib'(k-1, y, x+y), if k > 0
```

Checking to see if an application has been evaluated previously is avoided, because we have ordered the evaluation so that at every point we know we have already done what’s needed

With LCS, the information that needs to be stored is more complicated…

Dynamic Programming for LCS

- Given X of length *m* and Y of length *n*, we need to build a table of applications LCS(X_i, Yj), for all $0 \le i \le m$ and $0 \le j \le n$
- The (i, j) entry holds two pieces of information:
  - the length of LCS(X_i, Yj)
  - an arrow denoting the rule used for that entry
- Consider X = 01101001 and Y = 110110
- We know all of the boundary cases
  - Whenever either string is empty, the LCS is empty
  - So the initial table looks like this
  - The yellow entry will hold the info for LCS(01101, 110)

X \ Y	—	1	1	0	1	1	0
—	0	0	0	0	0	0	0
0	0						
1	0						
1	0						
0	0						
1	0						
0	0						
0	0						
1	0						

Populating the table

Each entry depends on three other entries:

- The one to the left of it
- The one above it
- The one diagonally above-left of it

Therefore we can fill in the table one row at a time, working down the rows and going right along each row

- (Clearly we could do it column-wise instead)
- For each entry, use the LCS recurrence relation

X \ Y	—	1	1	0	1	1	0
—	0	0	0	0	0	0	0
0	0	↑ 0	↑ 0	\ 1	← 1	← 1	\ 1
1	0						
1	0						
0	0						
1	0						
0	0						
0	0						
1	0						

$$
\begin{aligned}
\text{len}(X_i, Y_j) &= 0, \text{ if } ij = 0 \\
&= \text{len}(X_{i-1}, Y_{j-1}) + 1, \text{ if } x_i = y_j \\
&= \text{len}(X_{i-1}, Y_j), \text{ if } x_i \neq y_j \ \&\ \text{len}(X_{i-1}, Y_j) \geq \text{len}(X_i, Y_{j-1}) \\
&= \text{len}(X_i, Y_{j-1}), \text{ if } x_i \neq y_j \ \&\ \text{len}(X_{i-1}, Y_j) \leq \text{len}(X_i, Y_{j-1})
\end{aligned}
$$

The final table

- The final table looks like this
  - When different rules would give the same number, we can use any of them
- The length of LCS(01101001, 110110) is given by
  the last (bottom-right) entry
- To extract the LCS, follow the arrows up to the top-left
  - Wherever we encounter a \, add that character
  - Thus the LCS is 11010, denoted by yellow entries
- Note that if we had chosen different arrows (when we could), we may have gotten a different LCS
  - e.g. if (8,6) was "← 5", what would the LCS be?
- The complexity now is O(*mn*)!

X \ Y	—	1	1	0	1	1	0
—	0	0	0	0	0	0	0
0	0	↑ 0	↑ 0	\ 1	← 1	← 1	\ 1
1	0	\ 1	\ 1	↑ 1	\ 2	\ 2	← 2
1	0	\ 1	\ 2	← 2	\ 2	\ 3	← 3
0	0	↑ 1	↑ 2	\ 3	← 3	↑ 3	\ 4
1	0	\ 1	\ 2	↑ 3	\ 4	\ 4	↑ 4
0	0	↑ 1	↑ 2	\ 3	↑ 4	↑ 4	\ 5
0	0	↑ 1	↑ 2	\ 3	↑ 4	↑ 4	\ 5
1	0	\ 1	\ 2	↑ 3	\ 4	\ 5	↑ 5

Next time: Optimisation!

The 0-1 knapsack problem

THE UNIVERSITY OF WESTERN AUSTRALIA

- Given a set of items *X*, each with a weight and a value, and given a knapsack *K* that can hold weight *w*, find $X' \subseteq X$ such that
  - the items in *X'* fit into *K* (weight-wise), and
  - the total value of the items in *X'* is maximised
- e.g. *K* might be your MP3 player with capacity *w*, and *X* might be the set of songs that you have, for each of which the weight represents its file-size, and the value represents how much you like it
- Whilst the 0-1 Knapsack problem is known to be NP-hard, the seemingly more-complicated Fractional Knapsack problem is trivial to solve with a greedy algorithm
  - In Fractional Knapsack, you can pack any fraction of any item into *w*
- We will examine a dynamic programming algorithm for 0-1 Knapsack that gives reasonable performance

Recursion for the 0-1 knapsack problem

THE UNIVERSITY OF WESTERN AUSTRALIA

- Remember the essential issues with dynamic programming are to
  - Express the problem as one or more recurrence relations
  - Organise the sub-problems so that repeated work is minimised or eliminated

- An instance of 0-1 Knapsack with n items has the form

$$(\{w_1, \ldots, w_n\}, \{v_1, \ldots, v_n\}, w)$$

- Consider solving sub-problems of the form

$$(\{w_1, \ldots, w_m\}, \{v_1, \ldots, v_m\}, w')$$

where $m \leq n$ and $w' \leq w$

- Let $V(m, w)$ be the value of the optimal solution where we choose from the first m items with capacity w

- Either the m^{th} item is packed or it isn't, so

$$V(m, w) = max(v_m + V(m - 1, w - w_m), V(m - 1, w))$$

- With the trivial base cases $V(0, w) = V(m, 0) = 0$, and incorporating checks for exceeding w, this gives a (very inefficient) recursive algorithm

Dynamic Programming for Knapsack

THE UNIVERSITY OF WESTERN AUSTRALIA

- We will construct a table in similar fashion to the longest common subsequence problem
- Consider the instance ({*1, 2, 3*}, {*2, 3, 4*}, *5*)
- Each entry in the table gives *V(m, w)*
- The first row and column come from the base case
- The other entries come from the recursive rule
  - Applied row-by-row left-to-right, as previously

$$V(m, w) = max(V(m - 1, w - w_m) + v_m, V(m - 1, w))$$

m \ *w*	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					

The final result…

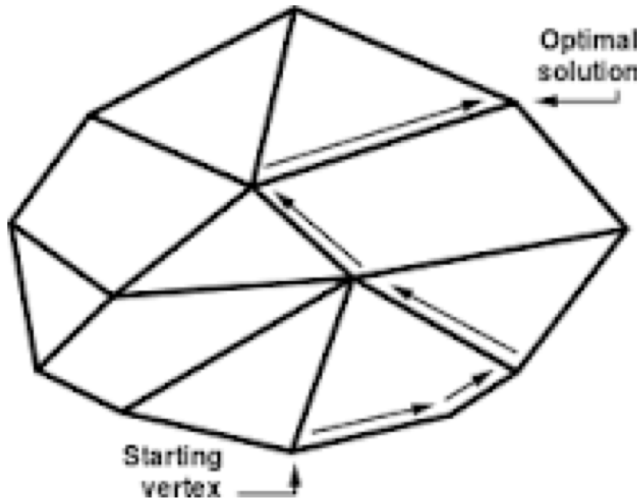
m \ *w*	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	2	2	2	2	2
2	0	2	3	5	5	5
3	0	2	3	5	6	7

- This algorithm is O(*nw*), because that's the size of the table
  - And clearly it gives the optimal solution in this case
- But be clear that this is ***not*** a polynomial-time algorithm for an NP-hard problem!
  - Why not?
- We can easily extract the items that form the solution
  - Working from the bottom-right, use the equation

$$T(m, w) = T(m-1, w), \text{ if } V(m, w) = V(m-1, w)$$
$$= \{m\} \cup T(m-1, w-w_m), \text{ otherwise}$$

Linear Programming Problems

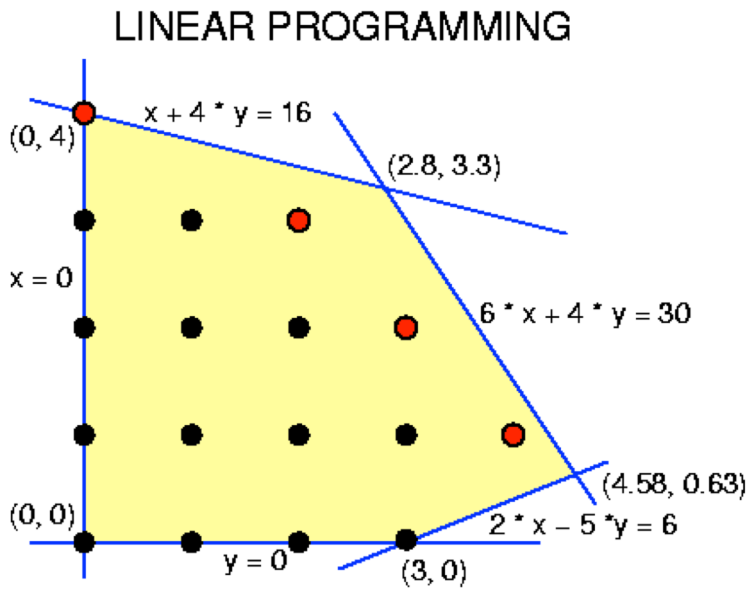
- Recall that Fractional Knapsack is the same as 0-1 Knapsack except that you can pack any fraction of any item
- This is an example of a *linear programming problem*, which is any problem of the form
  - Find real numbers $x_1, \ldots, x_n, x_i \geq 0$
  - That maximise $i=1ncixi$
  - Subject to the constraints $i=1naijxi<bj, 1 \leq j \leq m$
- A linear programming problem is characterised by
  - A cost vector $c_1, \ldots, c_n$
  - A bounds vector $b_1, \ldots, b_m$
  - An $n \times m$ array of constraint coefficients a_{ij}
- For Fractional Knapsack:
  - The values constitute the cost vector
  - The capacity is the only bound (i.e. $m = 1$)
  - The weights constitute the constraint coefficients



- All linear programming problems can be solved by the *simplex algorithm*, which has exponential complexity but is generally feasible in practice
  - Simplex is effectively a hill-climbing algorithm

Integer linear programming problems

- Adding the requirement that all solutions to a linear programming problem be integer values gives an *integer linear programming problem*
  - i.e. all of the x_i are required to be integers
  - if some solutions need to be integers, it's a *mixed integer linear programming problem*
- 0-1 Knapsack can be written as an integer linear programming problem, as can Travelling Salesman
- Given that both of these problems are NP-hard, as previously stated, we should not expect to find a feasible algorithm for solving integer linear programming problems
  - But if we did, that'd be great!
- Mixed integer linear programming problems are very common and there are some very good approximate solvers on the market, so often it is enough to translate a problem to a MILP.



Function to maximize: f(x, y) = 6 * x + 5 * y
Optimum LP solution (x, y) = (2.4, 3.4)
Pareto optima: (0, 4), (2, 3), (3, 2), (4, 1)
Optimum ILP solution (x, y) = (4, 1)

Approximation Algorithms

THE UNIVERSITY OF WESTERN AUSTRALIA

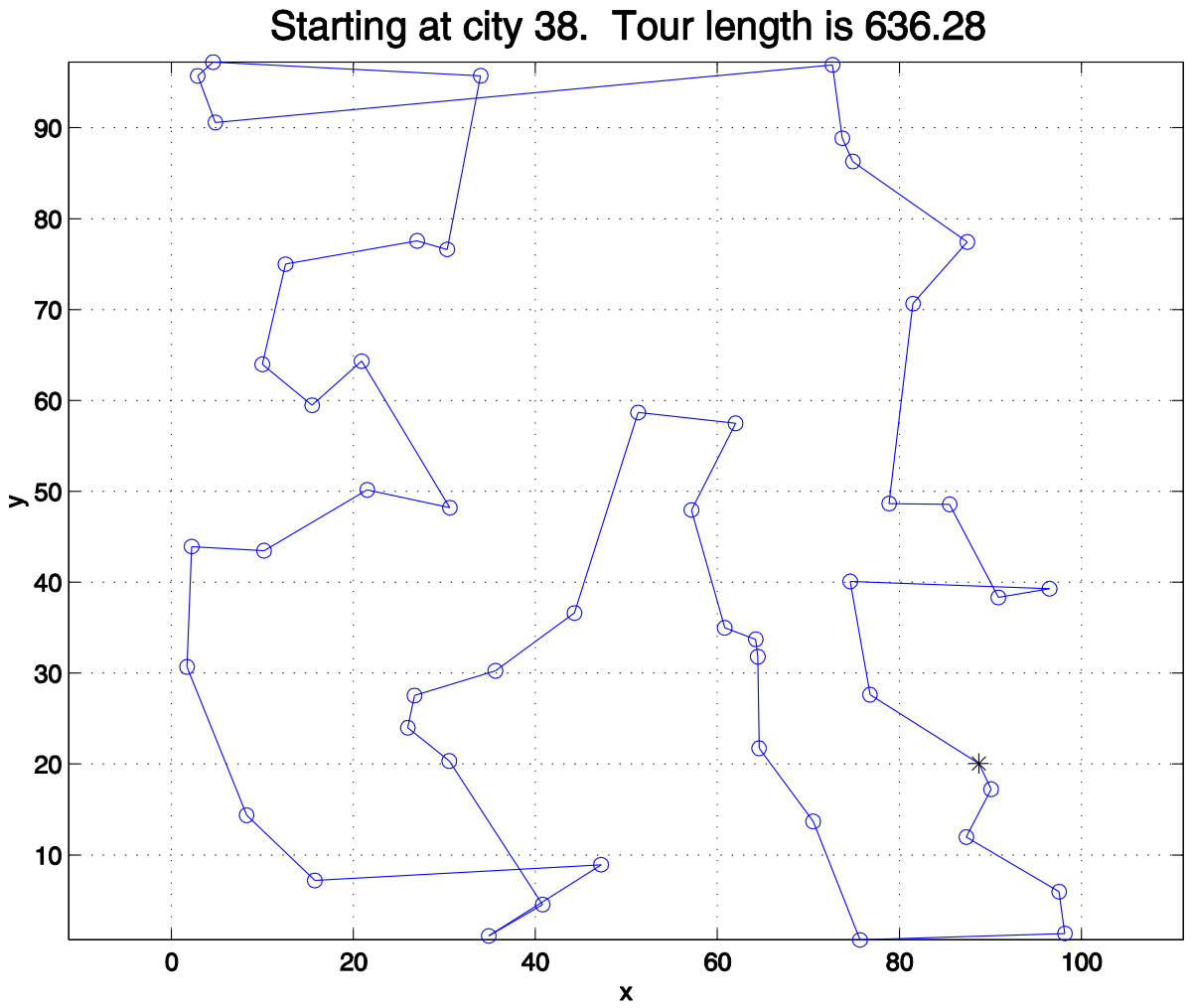
- An *approximation algorithm* is an algorithm that produces solutions to NP-hard problems
  - But with no guarantee that the solutions are optimal
- e.g. an approximation algorithm for Travelling Salesman would return some circular tour – hopefully a good tour, but not necessarily the best tour
- The performance of an approximation algorithm *A* on a given problem instance *I* is often described by the ratio *A(I)/OPT(I)*
  - *A(I)* is the value returned by *A*
  - *OPT(I)* is the optimum value (if known)
  - Sometimes a known bound is substituted for *OPT(I)*
- For many such problems standard benchmark data exists for comparing algorithms' performance
  - e.g. TSPLIB for Travelling Salesman

Greedy approximation for TSP

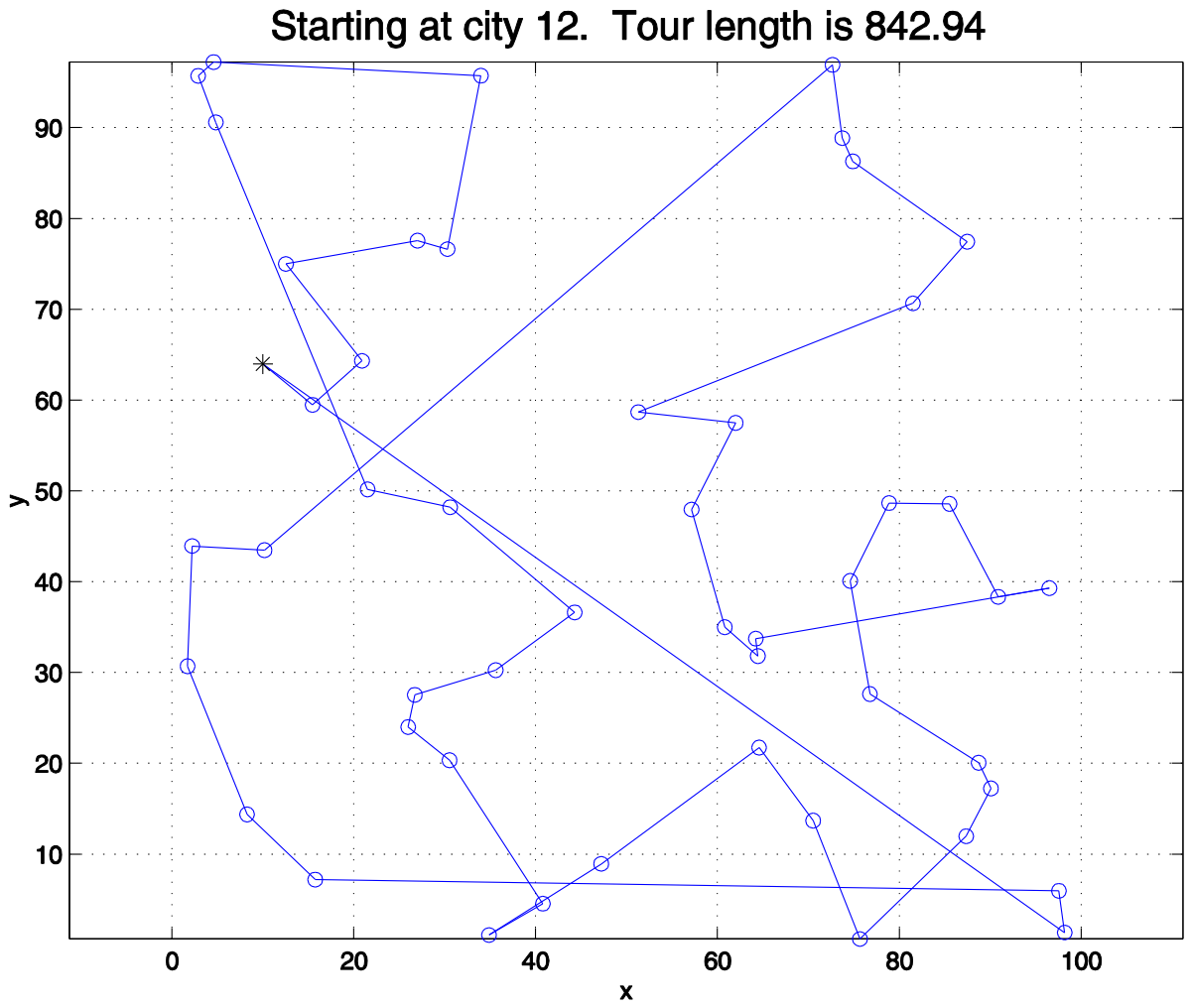
THE UNIVERSITY OF WESTERN AUSTRALIA

- Consider *geometric* instances of the TSP
  - The distance function is symmetric
    - $d(c_i, c_j) = d(c_j, c_i)$
  - And it satisfies the triangle inequality
    - $d(c_i, c_k) \leq d(c_i, c_j) + d(c_j, c_k)$
- It is easy to imagine non-geometric instances, but many approximation algorithms work well only for geometric
- One simple algorithm is *Nearest Neighbour* (NN)
  - Start at a randomly-chosen city
  - Always visit next the closest unvisited city
- NN is clearly $O(n^2)$
  - Unfortunately it is not very good!
- We could imagine a version of NN that just tries all possible starting cities
  - Obviously that would be $O(n^3)$

NN for a geometric TSP: best case



NN for a geometric TSP: worst case



TSP Theorems

THE UNIVERSITY OF WESTERN AUSTRALIA

- Theorem:

  For any constant $k > 1$ there are instances I of the TSP such that $NN(I) \geq k\ OPT(I)$

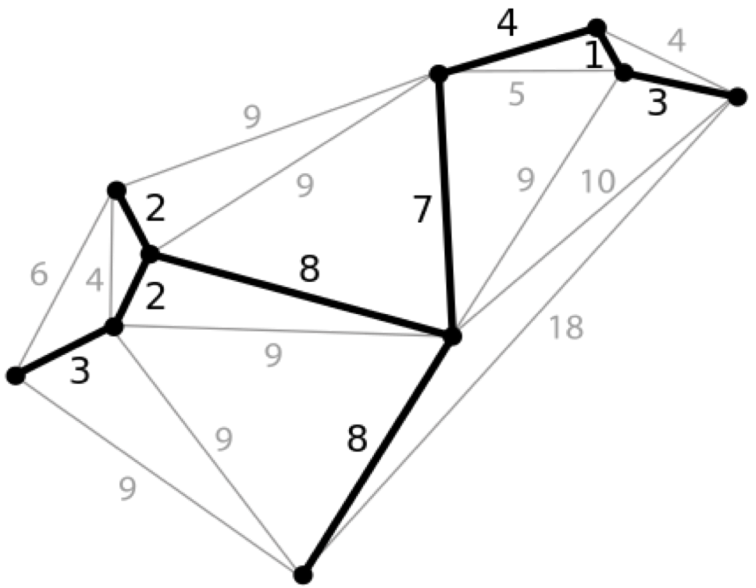
- Theorem: suppose A is a polynomial-time approximation algorithm for the TSP such that $A(I) \leq k\ OPT(I)$ for some constant k and for all instances I
  - Then there exists a polynomial-time algorithm to solve the TSP
  - Thus P = NP!
  - (Which most people don't believe…)

- Thus it seems hopeless to search for a decent approximation algorithm for the TSP…
  - But for geometric instances we can do better!

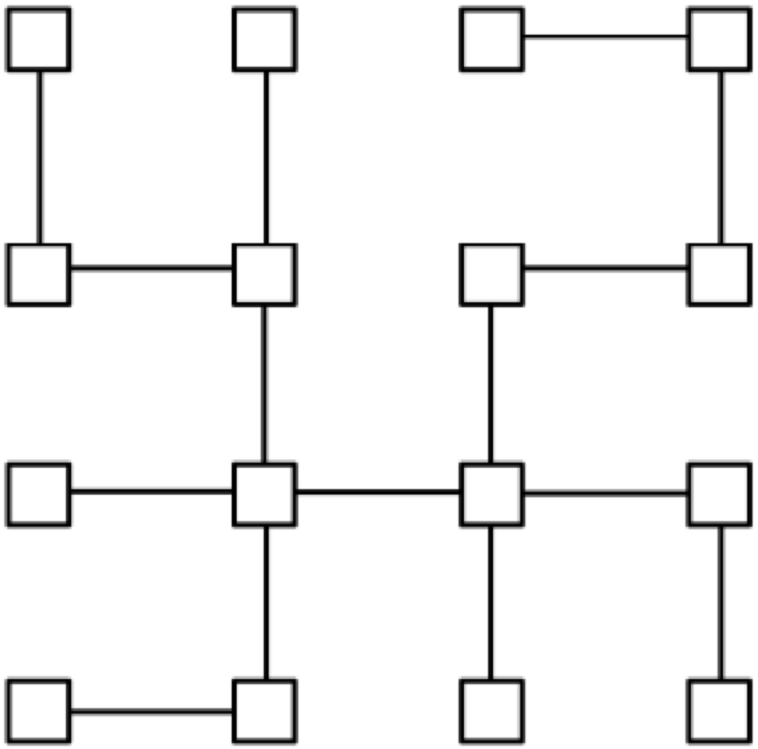
Minimum spanning trees for TSP

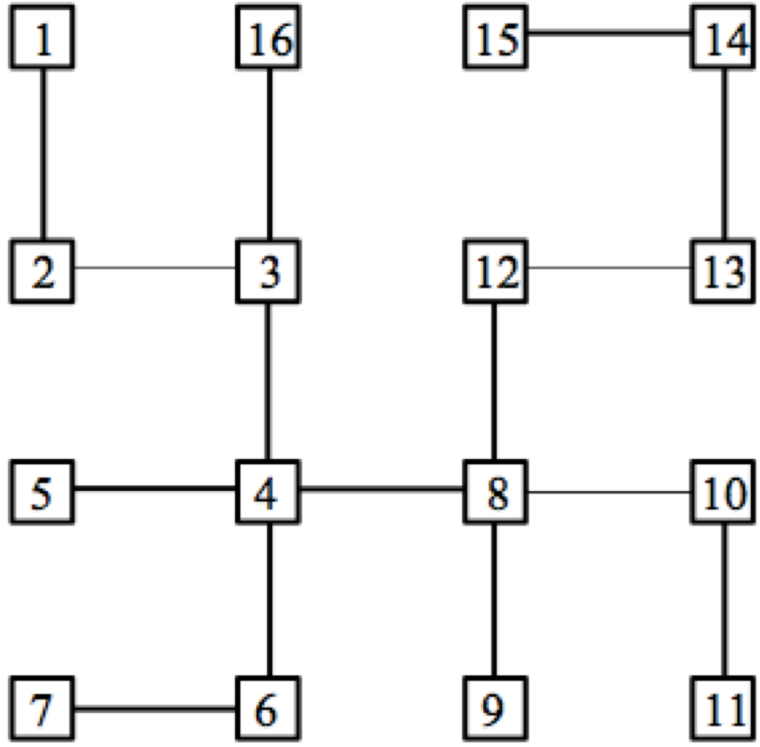
- The following algorithm is guaranteed to find a TSP tour for geometric *I* that is at most twice the optimal length
  - Find a minimum spanning tree for *I*
  - Do a depth-first search on the tree
  - Visit the vertices in order of discovery time

- Given a graph *G*, a *spanning tree* for *G* is a subset of *E(G)* that is a tree, and that connects all of *V(G)*
  - A *minimum spanning tree* for *G* is a spanning tree for *G* with the smallest possible weight
  - *G*'s MST can be found in O(|*E(G)*|) time



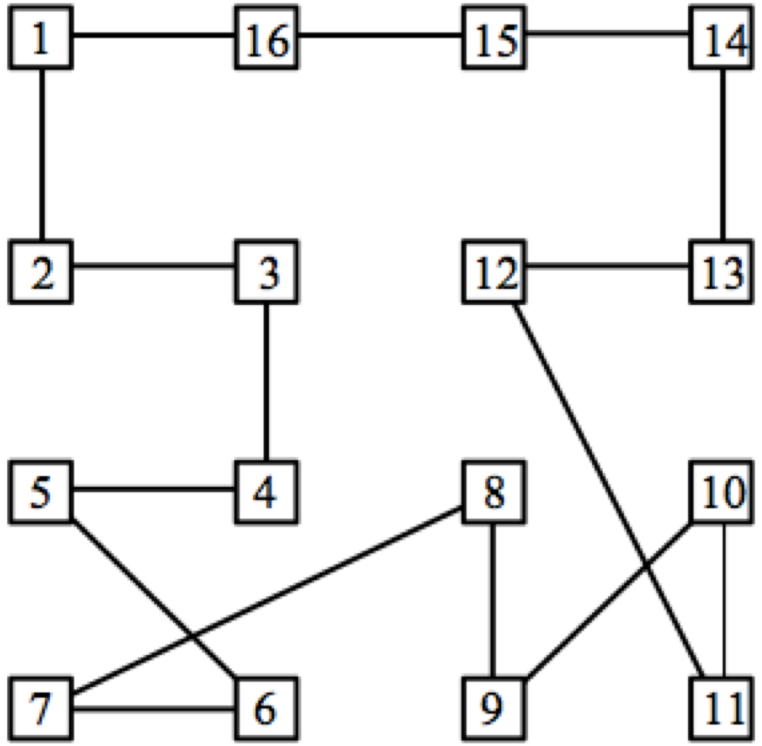
Create and search the MST





… and take short cuts

- If we simply follow the depth-first search – including the backtracking – we would walk along each edge once in each direction
  - This would create a tour that has length twice the MST, but with duplicated vertices
- The simplest solution is to take “shortcuts”, following the ordering of the vertices
  - i.e. visit them in order of discovery



Performance of MST-TSP

- The algorithm is guaranteed to find a TSP tour for geometric *I* that is at most twice the optimal length

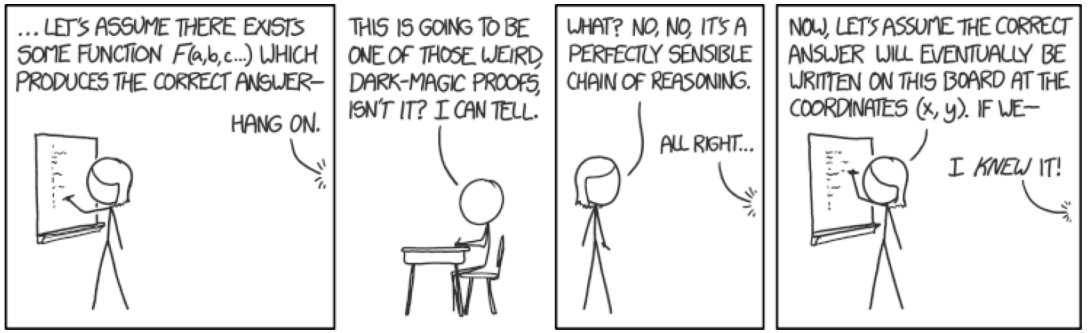
- Observe first that removing one edge from the optimal tour for *I* gives a spanning tree for *I*

OPT(I) – <one edge> = SpT

∴ OPT(I) – <one edge> ≥ MSpT

∴ MSpT < OPT(I)

∴ 2 MSpT < 2 OPT(I)



A(I) ≤ 2 MSpT

∴ A(I) < 2 OPT(I)

Still, a factor of 2 (100% error) isn't great. Can we do better on typical problems?

Insertion Algorithms for the TSP

THE UNIVERSITY OF WESTERN AUSTRALIA

- Insertion algorithms maintain a cycle through a subset of vertices, and insert new vertices into this cycle.

- At each stage we apply an *insertion method M* that inserts one vertex into a closed tour *C*
  - *M* selects an unused vertex *x*, then it inserts *x* into *C* at its *best position*
- To determine the best position, we consider each edge $(u, v) \in C$, and we select the edge that minimises

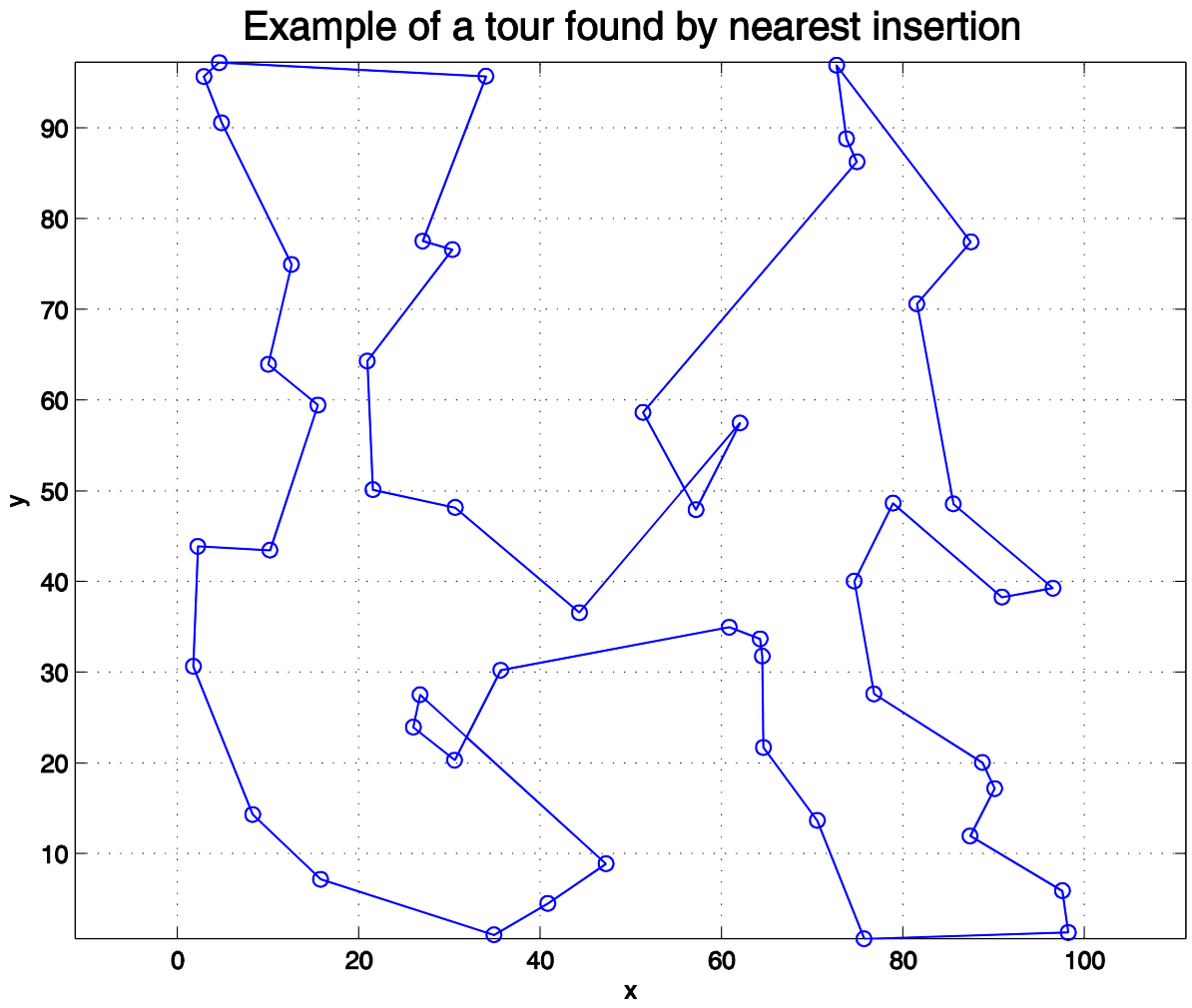
$$d(u, x) + d(x, v) - d(u, v)$$

- Then (u, v) is deleted, and (u, x) and (x, v) are added, creating a tour with one extra city

- Three common insertion methods are:
  - *Nearest insertion*: choose the *x* closest to *C*
  - *Farthest insertion*: choose the *x* furthest from *C*
  - *Random insertion*: choose *x* randomly

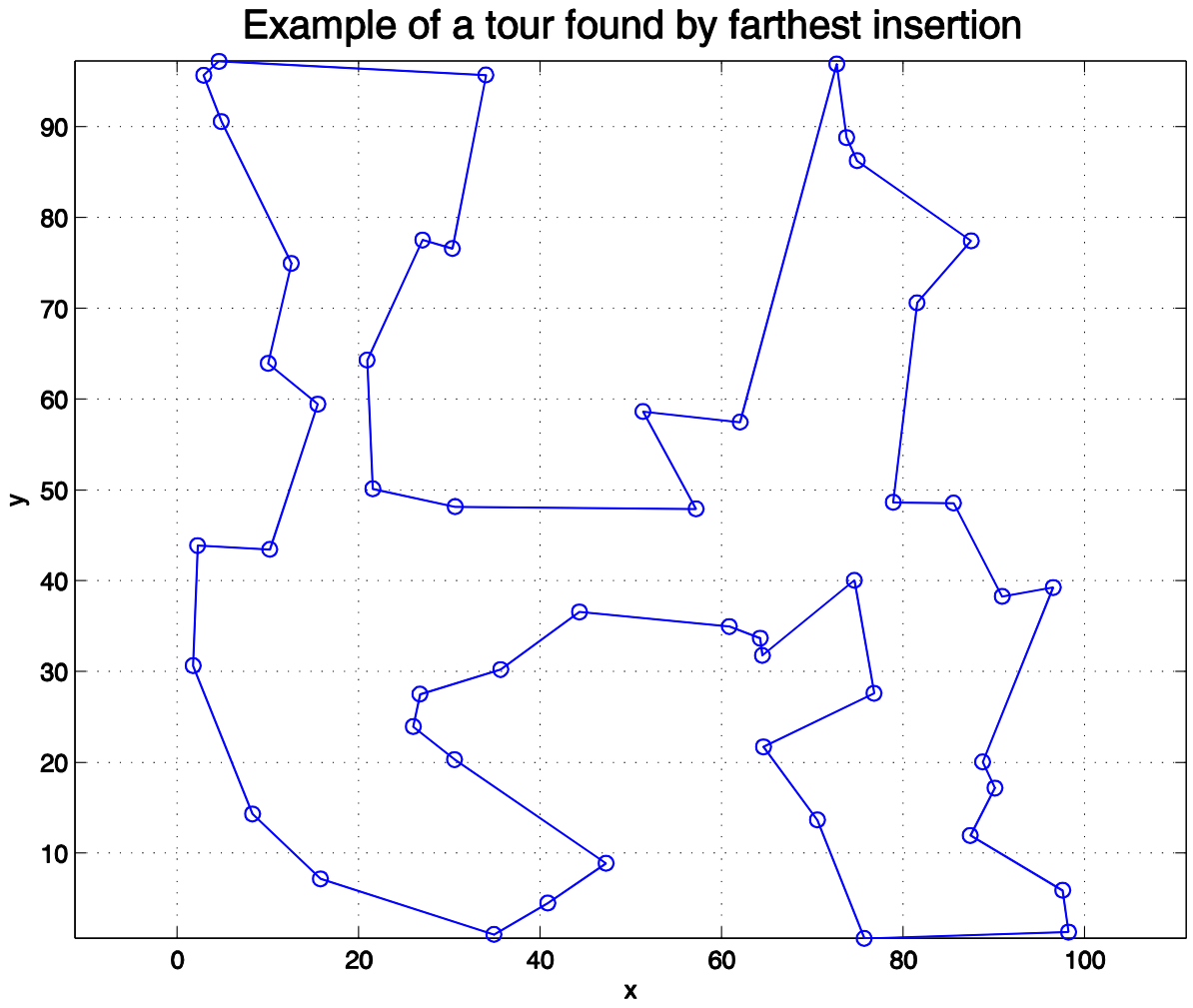
Tours found by nearest insertion

- ranged from 631-701 in length



Tours found by farthest insertion

- Ranged from 594-679 in length



Tours found by random insertion

- ranged from 607-667 in length

Cheapest insertion

THE UNIVERSITY OF WESTERN AUSTRALIA

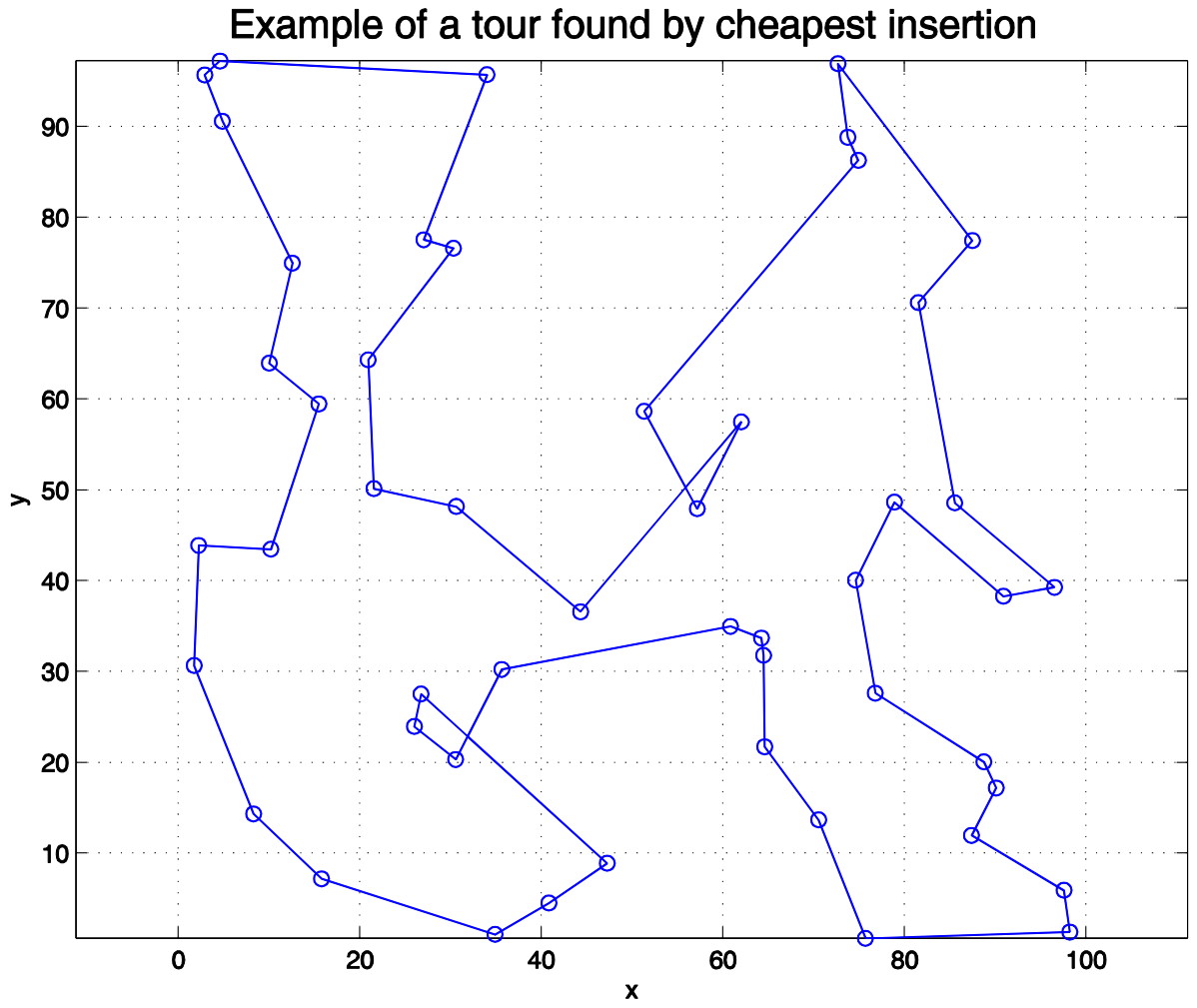
- With this method we search through all edges $(u, v) \in C$ *and* all vertices $x \notin C$ to find the pair that minimises

$$d(u, x) + d(x, v) - d(u, v)$$

- The previous three methods work in $O(n^2)$ time
  - Because they separate choosing a vertex from choosing an insertion point
  - Cheapest insertion seems to require at least an additional factor of *logn*

- The nearest insertion and cheapest insertion methods can be shown to produce tours of length no greater than twice the optimum
  - They are related to MST algorithms

Example of cheapest insertion

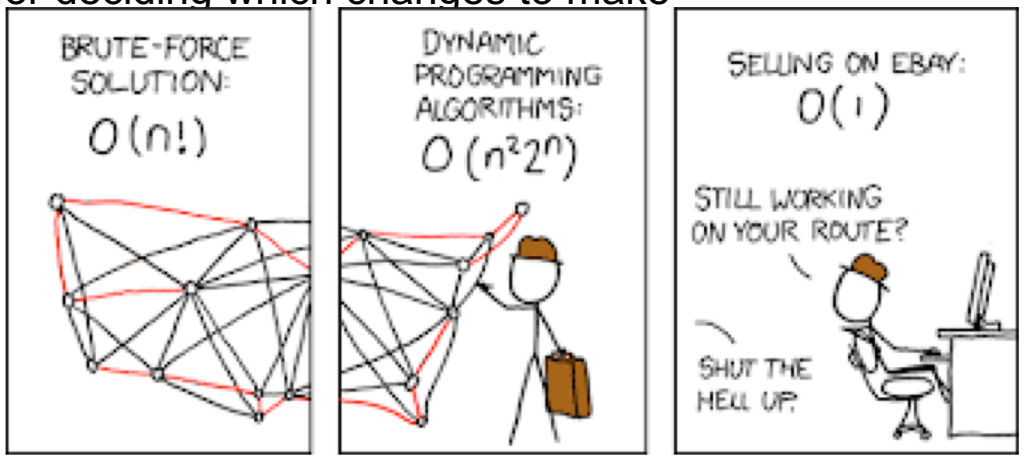


Iterative improvement

- One common feature of the tours produced by the greedy heuristics is that it is usually easy to see how they can be improved by changing a few edges here and there

- This leads to the idea of *iterative improvement*
  - Create a feasible solution (quickly)
  - Modify it slightly (and repeatedly) to improve it

- An iterative improvement algorithm requires
  - A *rule* for changing one feasible solution to another
  - A *schedule* for deciding which changes to make

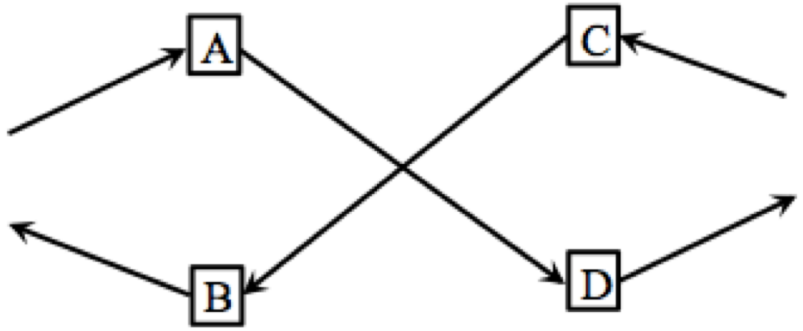


Improving TSP tours

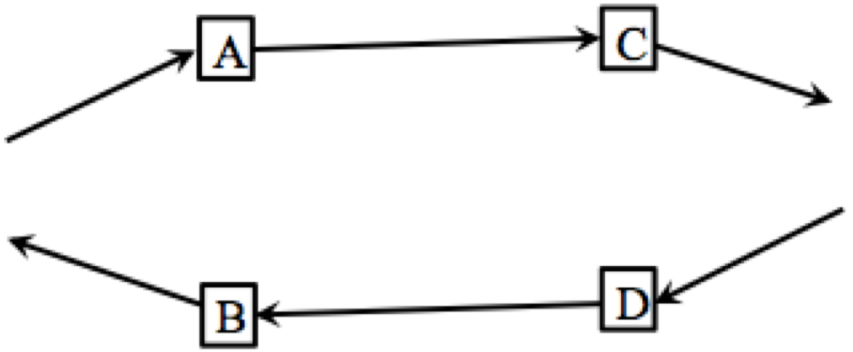
- A basic move for improving TSP tours involves deleting two edges and replacing them with two non-edges

- Consider the tour

  $A \rightarrow D \rightarrow \ldots \rightarrow C \rightarrow B \rightarrow \ldots \rightarrow A$



- *AD* and *CB* can be replaced by *AC* and *DB*



It should be clear that the result of this is still a valid tour

  – And that $D \rightarrow \ldots \rightarrow C$ is reversed

2-OPT

THE UNIVERSITY OF WESTERN AUSTRALIA

- Consider an iterative improvement algorithm that, in every iteration, examines every pair of edges in a tour, and performs an exchange if it would improve the tour
- This procedure must eventually terminate
  - The resulting tour is called *2-optimal*

- More complicated schemes involve deleting three edges and reconnecting the tour
  - Or in general *k* edges

- A tour that cannot be improved by a *k*-edge exchange is called *k-optimal*
  - In practice it is unusual to go beyond 3-optimal, because of the expense and because of diminishing returns
  - Note that this only applies for geometric instances

A state space graph

THE UNIVERSITY OF WESTERN AUSTRALIA

- We can abstract this process to consider a heuristic search on a huge graph called the *state-space graph* or the *search space* of the problem

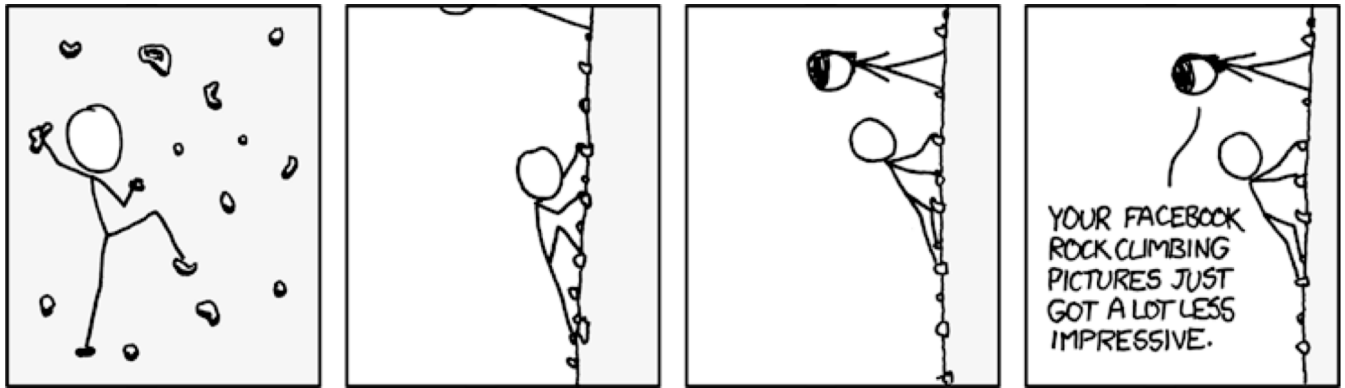
- The state-space graph of an instance *I* of the TSP is denoted by *S(I)*
  - The vertices of *S(I)* comprise all feasible tours for *I*
  - Two vertices in *S(I)* are connected iff they can be obtained from each other by the edge-exchange procedure described previously

- Each vertex *T* of *S(I)* has an associated *cost c(T)* which is the length of the tour *T*

- To completely solve the instance *I* requires finding which of the *(n–1)!* vertices of *S(I)* has the lowest cost

Gradient-based search

- Needless to say *S(I)* is usually VAST!
- But note that
  - A greedy insertion method returns one vertex in *S(I)*, i.e. one tour
  - An iterative improvement heuristic allows us to "walk" through *S(I)* by moving from one tour to its neighbours
- Conceptually we have a "current" tour *T*, and in each iteration
  - We generate a neighbour *T'* of *T*, and
  - We decide whether or not to "move" to *T'*
- This is the basis of *gradient-based search* algorithms
- The simplest gradient-based search procedure is known as *hill-climbing*
- Systematically generate neighbours *T'* of the current tour *T* and move to the first one with lower cost than *T*
- The process terminates when it is at a *T* that has no neighbours of lower cost
  - At this point *T* is 2-optimal
- An obvious variant (there are always variants!) is to always choose the *best* move at each step
  - Greedy iterative improvement!



Local optima

THE UNIVERSITY OF WESTERN AUSTRALIA

- A hill-climb will always finish at a vertex which has a lower cost than its neighbours
  - Such a vertex is a *local minimum* or *local optimum* of the state-space
- Unfortunately the state-space graph has an enormous number of local optima, only one of which is the global optimum that we would like to reach
  - Or sometimes there are multiple global optima…
- If we picture the search space as a kind of landscape where the height of a vertex corresponds to its cost,
  then *S(I)* is a *savagely* jagged landscape of enormously high dimension
  - Hill-climbing has no way to avoid the local optima in this landscape
  - Because all moves are local improvements
-
- Many techniques have been proposed which incorporate into gradient search some way of escaping local optima
  - It is an area of very active research
- We will mention three techniques
  - Simulated annealing
  - Tabu search
  - Genetic algorithms

Simulated annealing

THE UNIVERSITY OF WESTERN AUSTRALIA

- Simulated annealing tries to avoid local optima by allowing the search process to take "backward" moves
- Conceptually, it is based on the physical process of annealing, by which some solids can form crystals during cooling, if they cool slowly enough
  - The solid "searches for" the molecular configuration with the lowest potential energy
- Each iteration takes the form
  - Randomly generate a neighbour *T'* of the current *T*
  - If $c(T') \leq c(T)$, accept *T'*
  - If $c(T') > c(T)$, accept *T'* **with probability** *p*
  - So now "backward moves" are allowed and we can escape local optima
- But of course we don't want to make backward moves near the end of the process
  - Backward move are intended to move us to a new part of the space, so we can search there
  - So we dynamically alter *p* to make backward moves less likely as time goes on
- We maintain a *temperature variable t* which goes down as time advances, and we relate *p* to *t*
- Simulated annealing has worked well with many optimisation problems, but the performance is highly problem-specific and dependent on the cooling schedule

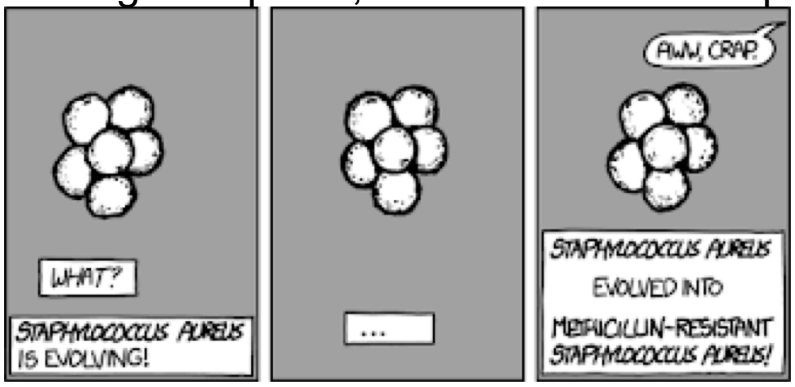
Tabu search

THE UNIVERSITY OF WESTERN AUSTRALIA

- The word *tabu* (or *taboo*) means something that is prohibited or forbidden
- *Tabu search* tries to avoid two weaknesses
  - The inability of hill-climbing to escape local optima
  - The early randomness of simulated annealing
- The aim is to spend most of the time exploring (near) local optima, whilst retaining the ability to escape them
- The fundamental idea is that we maintain a *tabu list* detailing the last *h* vertices that have been visited, and at each iteration
  - Select the best possible neighbour *T'* of *T*
  - If *T'* is not on the tabu list, move to *T'* and update the list
- Tabu search is very aggressive – each attempted move is in the best possible direction
  - Cycling would be inevitable without the tabu list
- The tabu list prevents the search from spending too much time near one local optimum, forcing it to visit other parts of the search space
- Tabu search also has been very successful, e.g. with football pools
  - But it is expensive and tricky to implement
  - Again results and settings (e.g. *h*) are highly problem-specific

Genetic algorithms

- *Genetic algorithms* (GAs) try to avoid getting stuck in local optima by maintaining a *population* of solutions
- The expectation is that these solutions will usually be distributed across the search space
  - It is highly unlikely that one (or even a few) local optima can trap all of them
- At each *generation* (iteration), the population of size *n* is used to create *n* new solutions, and the best *n* of these "survives" into the next generation
  - Again the conceptual inspiration is from the physical world, in this case the principle of evolution by natural selection
  - As well as being perturbed locally, solutions are combined using *crossover* in the hope of finding improved solutions
- GAs have been extremely successful on a wide range of problems, and they work well in difficult domains that often are beyond simpler methods
  - But again much fine-tuning is required, and often much computational power too!



Thanks!

THE UNIVERSITY OF WESTERN AUSTRALIA

- Next up…. agents!