Computational Methods for Tree Search

Michael Charleston

University of Tasmania
michael.charleston@utas.edu.au

2015.11.17

Really big numbers

n	# trees	*yes, but how much is that really?*
3	3	enumerable by hand
4	15	enumerable by hand
5	105	enumerable by hand on a rainy day
6	945	enumerable by computer
7	10395	still searchable very quickly on computer
8	135135	a bit more than the number of hairs on your head
9	2027025	population of Sydney living west of Paramatta
10	34459425	≈ upper limit for exhaustive searching; about the number of possible combinations of numbers in the National Lottery
20	8.2×10^{21}	≈ upper limit for branch-and-bound searching
48	3.21×10^{70}	≈ number of particles in the universe
136	2.11×10^{267}	number of trees to choose from in the "Out of Africa" data[1]

[1] Vigilant *et al.*., 1991

Which is the best tree?

Given the huge number of possible trees even for small data sets, we have two options:

- Build one according to some *algorithm*;
- Assign a "goodness of fit" criterion and *search* for the tree(s) which optimise(s) this criterion.

Building Trees From Scratch

The simplest way to build trees is by *construction*:
Most trees are constructed with an input matrix of distances.

What data should we use?

There are many types of data available to infer phylogenies:

- morphological characters
- morphometric
- DNA×DNA hybridization
- immunological distances
- molecular sequences
- RFLP (Restriction fragment length polymorphism)
- RAPD (Random ampliflied polymorphic DNA)
- SNP (Single nucleotide polymorphism)...

Distance Data

These data can be obtained in many ways, most of which do not have an explicit model, so it is difficult to correct for parallel or convergent evolution.

However, sequence data can be converted to distances, that *do* correspond to a model — by correcting for multiple hits and other complexities (see later).

Constructive Algorithm

Algorithm 1: Construct-A-Tree(D) :T

given D, an $n \times n$ distance matrix
let n **be** the number of leaves in the final tree
let T **be** a tree // we will return this
$T \leftarrow \emptyset$ // T "gets" the value "empty set"
let b **be** the number of leaves we've built into the tree
$b \leftarrow n$
while $(b > 1)$ **do**
· Using distance matrix D, choose x, y to connect
· Create new node z as their most recent common ancestor (w.r.t. the others)
· Add edges (z,x) and (z,y) to T
· Calculate $d(z,w)$ for every remaining vertex w
· Remove rows and columns for x and y from D
· Insert new row and column for the $d(z,w)$'s
· $b \leftarrow b - 1$
return(T)

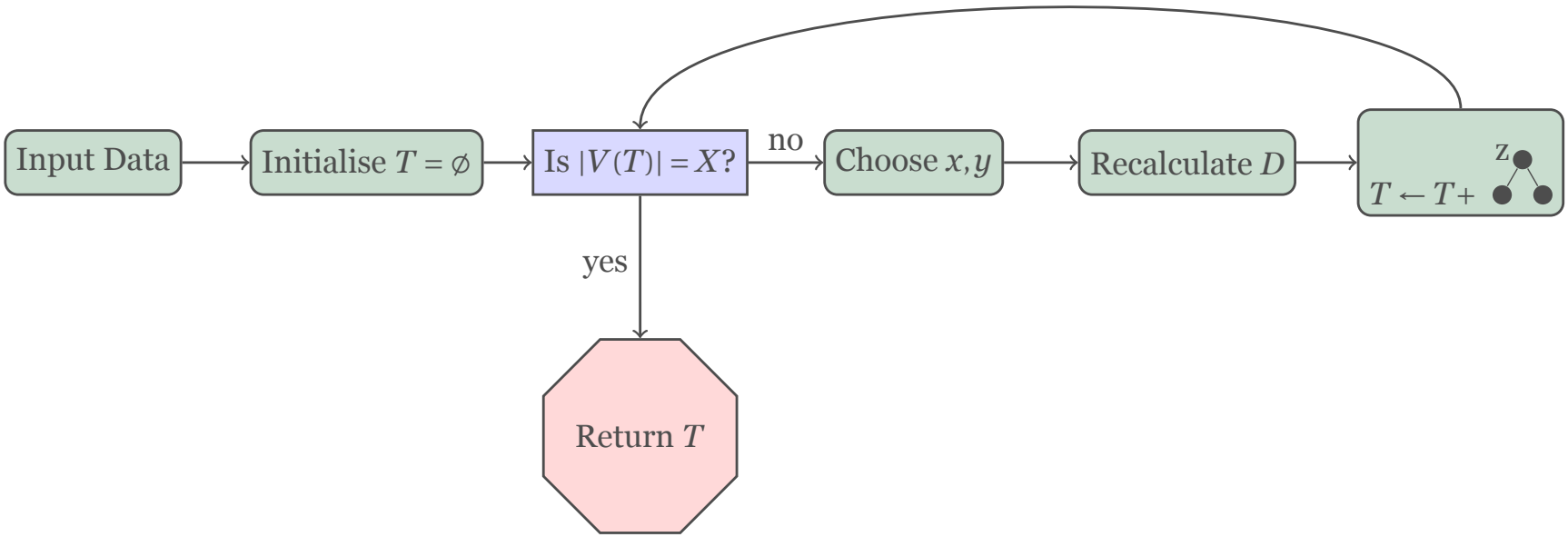
Constructive Algorithm (cont.)

Note that:

- We never go back to fix mistakes;
- We *always get a tree*;
- We don't get a sense of how good any other trees are;
- We don't get a sense of how "tree-like" the distances are.

Constructing a tree

Building a tree



Construction is FAST

As a kind of hand-wavy method of working out how fast the algorithm to construct a tree is, we make some simplifying assumptions (that turn out to be ok):

- We care about how the time taken to run the algorithm *grows* with the size of the input data;
- We ignore implementation details (programming language, computer);
- We ignore constant multipliers: to us, $45n^3$ is equivalent to n^3.
- Only the biggest (fastest growing) terms matter:
  $n^4 + \underbrace{1100^6}_{\text{a constant}} + \log(n) + n!$ is dominated by $\boxed{n!}$.

Construction is FAST (cont.)

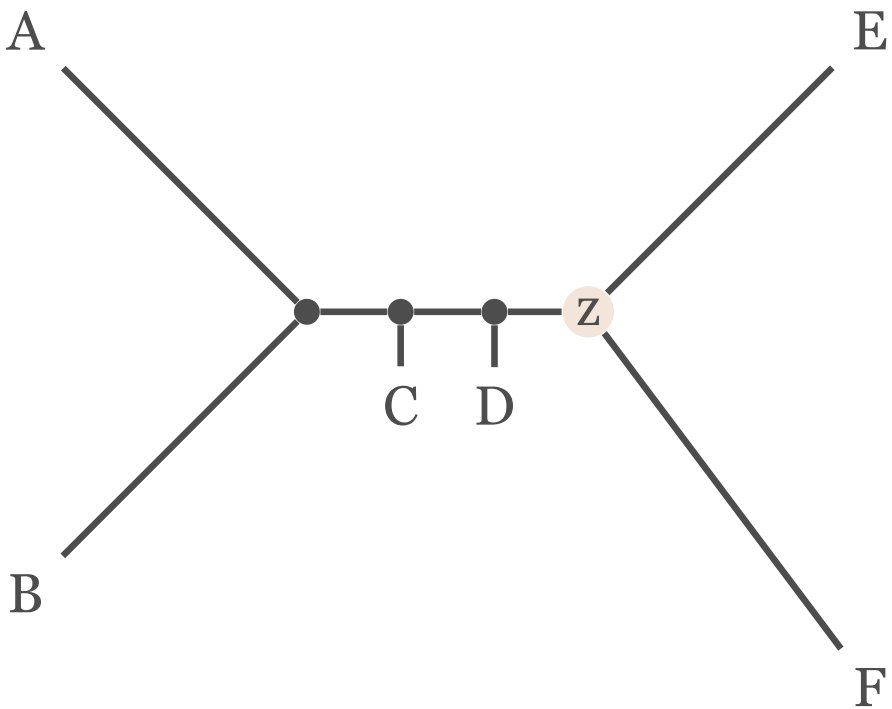
Construct-A-Tree(D) :T

1 **given** D, an $n \times n$ distance matrix
2 **let** n **be** the number of leaves in the final tree
3 **let** T **be** a tree // we will return this
4 $T \leftarrow \emptyset$ // T "gets" the value "empty set"
5 **let** b **be** the number of leaves we've built into the tree
6 $b \leftarrow n$
7 **while** $(b > 1)$ **do** // about n iterations
8 · Choose x, y from D // around $O(n^2)$
9 · Create new node z // trivial
10 · Add edges (z,x) and (z,y) to T // trivial
11 · Calculate $d(z,w)$ for every remaining vertex w // $O(n)$
12 · Remove rows and columns for x and y // trivial
13 · Insert new row and column // trivial
14 · $b \leftarrow b - 1$ // *really* trivial
15 **return**(T)

// $\boxed{trivia + n \times (O(n^2) + O(n) + trivia) = O(n^3)}$

Choosing x, y

One way is to choose x, y such that they are as close as possible to each other, but that can fail badly:



- Here, $d(C, D) = 3$ but they aren't each other's "closest" relatives.
- We should have joined (E, F) together first and formed z, their common ancestor;
- Then D and z could have been joined.

Neighbour Joining

Neighbo[u]r Joining uses "net divergence", a measure of how far each potential pair of leaves is, to choose which ones to join.
Given a matrix of distances D we use d_{ij} as a short-hand for the distance between leaves i and j.
The *net divergence* r_i of i is given by

$$r_i = \sum_{j \neq i} d_{ij},$$

that is, the sum of the distance of i to any other leaf.
NJ chooses which leaves to join by finding the minimum entry in matrix M, defined in turn by

$$M_{ij} = d_{ij} - \frac{1}{n-2}(r_i + r_j)$$

Example of NJ

Suppose we have

$$D = \begin{array}{c} \\ B \\ C \\ D \\ E \\ F \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{bmatrix} 5 & & & & \\ 4 & 7 & & & \\ 7 & 10 & 7 & & \\ 6 & 9 & 6 & 5 & \\ 8 & 11 & 8 & 9 & 8 \end{bmatrix} \end{array}$$

with $n = 6$. First, r_i's (without the gory details):

$$r_A = 5 + 4 + 7 + 6 + 8 = 30$$

$$r_B = 42, \quad r_C = 32, \quad r_D = 38, \quad r_E = 34, \quad r_F = 44$$

Example of NJ (cont.)

This gives us our M (recall $M_{ij} = d_{ij} - \frac{1}{n-2}(r_i + r_j)$):

$$M = \begin{array}{c} \\ B \\ C \\ D \\ E \\ F \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{bmatrix} \boxed{-13} & & & & \\ -11.5 & -11.5 & & & \\ -10 & -10 & -10.5 & & \\ -10 & -10 & -10.5 & \boxed{-13} & \\ -10.5 & -10.5 & -11 & -11.5 & -11.5 \end{bmatrix} \end{array}$$

The minimal entries are M_{AB} and M_{DE}; we have to choose one so we'll pick A, B and make new node z.

Example of NJ (cont.)

Calculating d_{zi} next we get:

$$d_{zC} = (d_{AC} + d_{BC} - d_{AB})/2 = 3$$
$$d_{zD} = (d_{AD} + d_{BD} - d_{AB})/2 = 6$$
$$d_{zE} = (d_{AE} + d_{BE} - d_{AB})/2 = 5$$
$$d_{zF} = (d_{AF} + a_{BF} - d_{AB})/2 = 7$$

— and we also can work out the branch lengths between z and A, B; call these w_{zA}, w_{zB}:

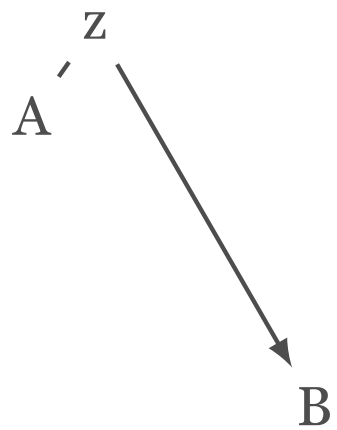
$$w_{zA} = \frac{d_{AB}}{2} + \frac{(r_A - r_B)}{2(n-2)} = 1; w_{zB} = d_{AB} - w_{zA} = 4$$

Example of NJ (cont.)

We now have a new distance matrix:

$$D' = \begin{array}{c} \\ C \\ D \\ E \\ F \end{array} \begin{array}{c} \begin{array}{cccc} z & C & D & E \end{array} \\ \begin{bmatrix} 3 & & & \\ 6 & 7 & & \\ 5 & 6 & 5 & \\ 7 & 8 & 9 & 8 \end{bmatrix} \end{array}$$

and we've begun the tree:



Searching for a tree

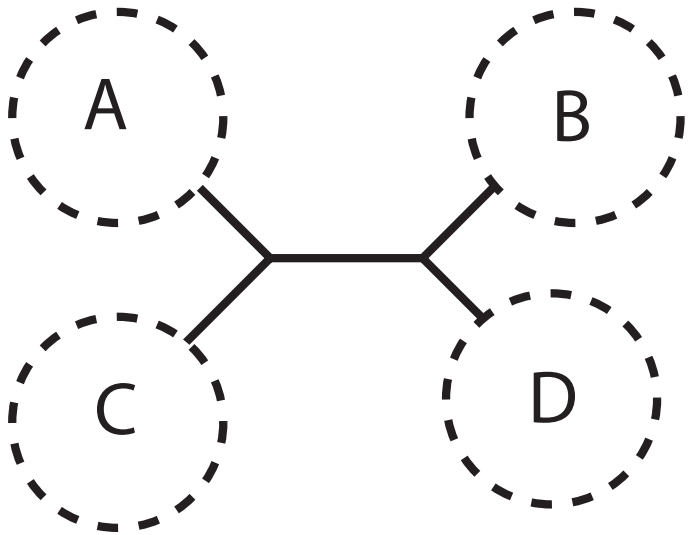
There are two ways in which we can search through tree space to find the best tree for our data:

- Branch-and-bound: finds the optimal tree by implicitly checking all possible trees
- Heuristic: searches by randomly perturbing the tree, does not check all trees and cannot guarantee to find the optimal one(s).

(we do not count exhaustive searching here, which is only possible for very small data sets)

Branch-and-bound 1

Effectively checks all possible trees but doesn't list them all.
Finds bounds on the best possible score for any tree containing just a given part:



In the above we don't care about the internal structure of the subtrees A, B, C, or D; just the relationship they have to each other.

Branch-and-bound 2

Parsimony is fine:

- adding more branches to a subtree can never decrease the tree length, so
- we can use a bound of the parsimony length of the subtree.

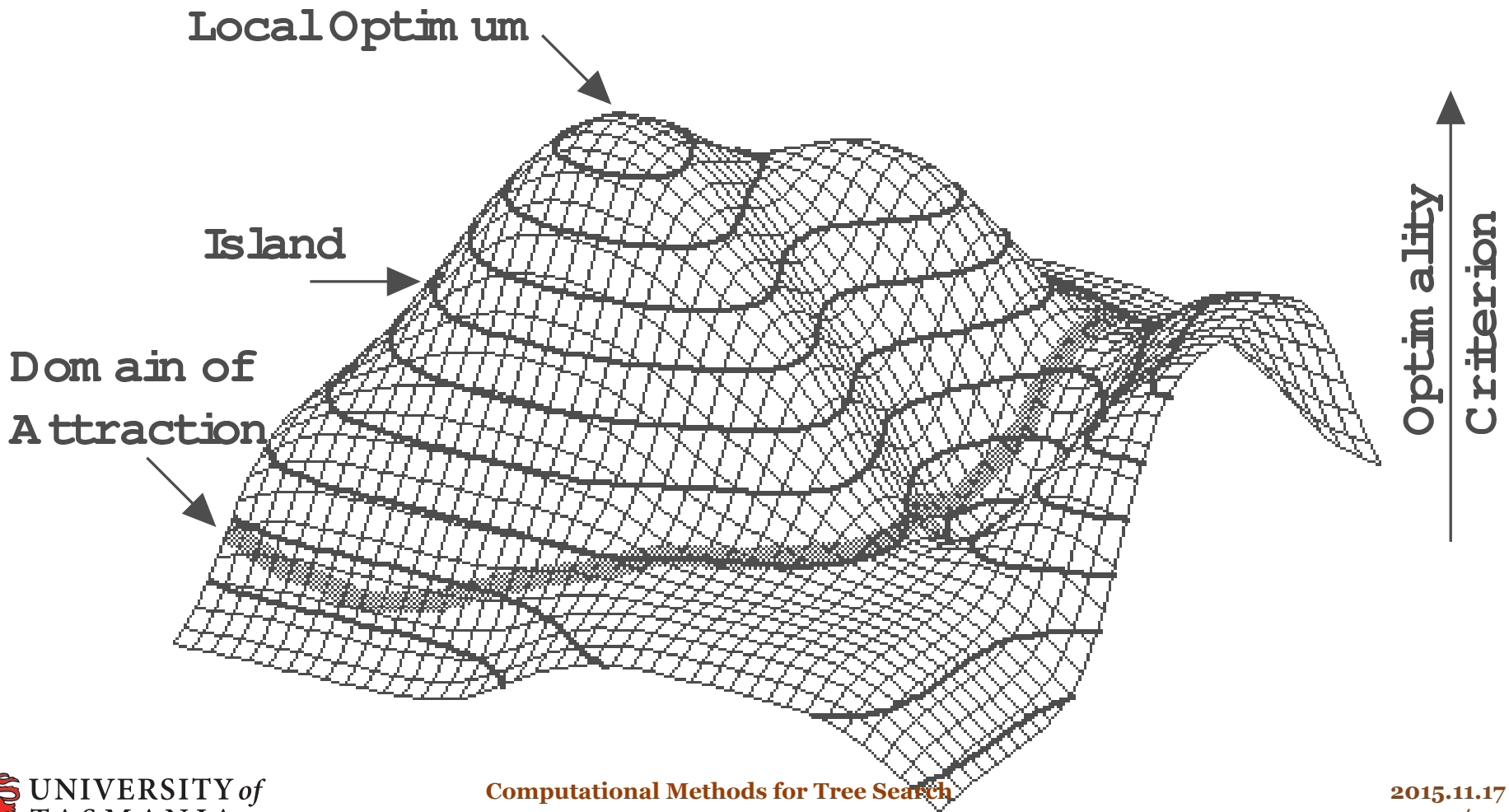
Heuristic Search

Heuristic methods do not offer any guarantees: they are the best we can do under difficult computational circumstances.

They can do very well but remember optimality is not guaranteed!

Landscapes

The landscape of the problem affects how well our heuristic searching will do.



Heuristic search in tree space

The process is quite simple.

1. Begin with a tree T.
2. Perturb T at random to get T' — this is the “branch-swapping” part of PAUP*.
3. Calculate the value of the optimality criterion of this tree.
4. If the new tree is acceptable, set T' to T and continue the random walk.
5. If not, keep trying different perturbations until we get bored.

Pros and Cons of heuristic search

Pros:

- It's *fast*, and is the only way to tackle large problems;

Cons:

- Cannot guarantee to find the optimal solution;
- Different runs can find different "best" trees;
- Sensitivity to search parameters.