INTRODUCTION

CITS4404
Artificial Intelligence & Adaptive Systems
Algorithms

• Algorithms are central to Computer Science
• Algorithms come in many forms and flavours
• The most common class of algorithm is what we might broadly call a “constructive algorithm”
• Given a problem, follow these steps and the problem will be solved!
An example constructive algorithm

- Mergesort is a classic example of a constructive algorithm
- Given a list of values $L$, perform these steps and $L'$ will be a sorted copy of $L$

1. Split $L$ down the middle such that $L = F + B$
2. Sort $F$ such that $F' = \text{sort}(F)$
3. Sort $B$ such that $B' = \text{sort}(B)$
4. Combine $F'$ and $B'$ to form the sorted list $L'$, by repeatedly taking the smaller of their heads
Behaviour of mergesort

Algorithm steps:
1. Divide the list into two halves: $L$ and $F$.
2. Recursively sort $F$ and $B$.
3. Merge $F$ and $B$ into $L'$.

Example:
- Initial list: $6 \ 8 \ 1 \ 15 \ 12 \ 2 \ 9 \ 7$
- Divided into $L$ and $F$:
  - $L$: $6 \ 8 \ 1 \ 15 \ 12 \ 2 \ 9 \ 7$
  - $F$: $1 \ 6 \ 8 \ 15$
- Divided into $B$ and $B'$:
  - $B$: $12 \ 2 \ 9 \ 7$
  - $B'$: $2 \ 7 \ 9 \ 12$
- Merge $F$ and $B$ into $L'$:
  - $F'$: $1 \ 6 \ 8 \ 15$
  - $B'$: $2 \ 7 \ 9 \ 12$
  - merged list: $1 \ 2 \ 6 \ 7 \ 8 \ 9 \ 12 \ 15$

Diagram:
- Arrows indicate the flow of data and operations.
Constructive algorithms are great!

- They always work
- They can be *proven* to work
- They have predictable performance
- Given several constructive algorithms to solve a given problem, we can select the best one for any given data
But they aren’t always available

- For some problems, there are no possible solutions
- For some problems, there are no known solutions
- For some problems, all known solutions are too expensive
- For some problems, we need to solve bigger instances than is currently possible
- For some problems in some contexts, resources are limited

- What can we do with such problems?
What features make constructive algorithms struggle?

- Complexity
- Size of interesting instances
- Missing information
- Uncertain information
- Noisy information
- Dynamic information
- Analogue information
What’s the alternative?

• The alternatives to constructive algorithms are all based in some way around the idea of searching
• If we don’t know how to directly solve a problem quickly, we instead generate solutions and then determine their value
• This is the idea of *enumeration*: generate all possible solutions, and either
  • Return the first “correct” solution
  • Return the “best” solution
• Note the distinction:
  • Constructively, we build a solution that we know will be correct
  • Enumeratively, we build solutions and *then* test them for correctness
Enhancing enumeration

• Enumeration is infeasible for complex problems: there are just too many possible solutions
• Backtracking makes enumeration faster by testing partial solutions and eliminating “failing” ideas early
• Depth-limited search limits the cost of the solutions searched for, and increases this cost until it’s successful
• A* ranks partial solutions on their estimated total cost, and tries promising ideas first
Abandoning optimality

- The approaches discussed in CIT4404 are intended for the most complex problems of all
  - For these kinds of problems, often we wouldn’t even recognise optimality if we saw it!
- They use simple mechanisms to find good solutions, rather than complex mechanisms to find optimal solutions
- They allow solutions to communicate and/or to build on one another, in order to generate good solutions quickly
- Many of these approaches are *anytime algorithms*
  - We work with complete solutions, and we can stop at any time
  - But the longer we search, the better we expect the result to be
Humans and other animals do amazingly well with such problems

• They can deal with huge complexity
  • Generalising from patterns, analogising, etc.
• They can abstract from detail
  • Classifying, using probabilities, etc.
• They don’t expect perfect answers
  • “good enough, soon enough, cheap enough”
• They don’t expect 100% success
  • Repeated attempts can help
• They can learn from experience
  • Both by trial-and-error, and more systematically
• We can use nature as a “proof of concept” for designing algorithms from a different perspective
Natural systems

• Four aspects of natural systems have been used as inspiration for algorithm design

• Evolution by natural selection
• Development of bodies and behaviour
• Learning how to thrive in an environment
• Cooperation between groups
Evolution – “designing the basics”

• Genes and chromosomes hold the “recipes” for nature’s “designs”
• DNA is the language in which these recipes are expressed
• Evolution, through the power of natural selection acting over immense geological time, provides the mechanism for reusing good designs, improving performance, and adapting designs to new environments
• Evolutionary techniques have been used widely to solve a huge range of problems in design and optimisation
Local projects

• Designing (networks of) equipment for the mining industry
• Designing torpedo tracks for the weapons industry
• Solving complex sports scheduling problems
• Solving complex 2D and 3D packing problems
• Evolving tactics for artificial soccer players
• Debugging logical errors in Java programs
• Many others…
Development – “connecting the components”

- The way that an organism develops and grows depends both on its internal structure – its genes – and on its external environment
  - e.g. your brain develops its “hardware” until at least your 20s, and possibly for much longer
  - e.g. scientists are trying to “recreate” extinct species by implanting ancient DNA in the eggs of closely-related extant species
- This interaction between genes and environment for development is one of the least-understood aspects of biology, and of our four examples it has had the least transfer to practical software systems
Local projects

- Developing topographical maps between the retina and the optic tectum in the brain
Learning – “training the software”

• Organisms are born with and develop many capabilities, but only through interaction with their environment do they learn how/when best to use these capabilities
  • *e.g.* a child learning not to touch a hot stove
  • *e.g.* a chess grandmaster, or a top surgeon

• It used to be blindly asserted that computers could never exhibit “intelligence” because they could only perform tasks that programmers had anticipated and created solutions for
  • How can I tell a machine to do a task that I can’t do?

• The field of “machine learning” has shown conclusively that this assertion is false
  • *e.g.* Deep Blue, Komodo, Stockfish (for chess)
Local projects

- Playing games (poker, Pac-Man, Go, others)
- Fraud detection
- Spam detection
- Identifying intention in natural language dialogue
- Tagging speech using case-based reasoning
- Planning induction
- Composing music
- Robot learning
Cooperation – “working together”

- The success of many species relies not just on the capabilities of individuals, but on the cooperative behaviour or social interactions of groups.
- Many species solve problems in groups that could never be solved by any single individual.
- Examples include ants, termites, bees, many predators, and humans.
- This has lead to many different types of algorithms, based on the behaviour of birds, ants, bees, markets, and others.
Local projects

• Deriving complex game strategies
• Optimising semi-conductor layouts
• Optimising weights in neural networks
• Solving classification problems
Key technologies

- Evolutionary algorithms
- Particle swarm optimisation
- Any colony optimisation
- Neural networks
- Learning classifier systems
- Artificial immune systems
- Fuzzy reasoning
- Bayesian reasoning
- Market-based algorithms

• Much more next week!