Data structures and algorithms, and Lisp

Adam Boult (www.bou.lt)

February 10, 2023

Contents

Preface

This is a live document, and is full of gaps, mistakes, typos etc.

Part I

Linked lists

Chapter 1

Linked lists

1.1 Single linked lists

1.1.1 Single linked lists

successor and key, null if nec

motivation. don't need to change who list to add things

1.1.2 Functions on linked lists

read, pop, add

1.2 Double linked lists

1.2.1 Doubly linked lists

successor, predecesor and key, null if nec

1.3 Sorting linked lists

1.3.1 Sorting linked lists

1.4 Write operations on arrays

1.4.1 The pop operation

1.4.2 The insert operation

Chapter 2

Insertion sort on linked lists

2.1 Insertion sort

2.1.1 Insertion sort on arrays

Insertion sorts can be more efficient with linked lists. Need to move less.

Part II

Using nodes as abstract data types

Chapter 3

Binary search trees and sets

3.1 Binary search trees

3.1.1 Introduction

3.2 Using binary search trees to implement sets

3.2.1 Introduction

Chapter 4

Heaps and heapsort

Part III

Solving tree problems

Chapter 5

Finding any path between nodes using stacks and depth-first search

5.1 Depth-first search

5.1.1 Depth-first search

A depth-first search operates Last-in First-out (LiFo). That is, it selects the newest frontier node. This results in a deep, rather than a broad search. Once the maximum depth has been reached, the algorithm will move towards breadth. Path cost is not considered in this algorithm.

May not find optimal solution, but is linear in space

Informed: No

Time: $O(b^m)$

Space: $O(bm)$

Complete: Yes

Optimal: No

5.2 Search algorithms

5.2.1 Search algorithms

A search algorithm takes a grid and identifies a path from a start point to an end point. Each node in the grid has connections to other nodes, with costs of

moving between nodes.

5.2.2 Types of nodes in a search algorithm

In each search algorithm there are three types of nodes: unexplored nodes, explored nodes and frontier nodes. At the start of the algorithm the start node is explored, each node connected to the start node is a frontier node, and all other nodes are unexplored nodes.

When an algorithm explores a frontier node, it is added to the explored nodes, and all new nodes are added to the frontier nodes.

Chapter 6

Finding the shortest path between nodes using queues and breadth-first-search

6.1 Breadth-first search

6.1.1 Breadth-first search

A breadth-first search operates First-in First-out (FiFo). That is, it selects the oldest frontier node. This results in a broad, rather than a deep search. Once all branches have been explored, the algorithm will move deeper. Path cost is not considered in this algorithm.

Informed: No

Time: $O(b^d)$

Space: $O(b^d)$

Complete: Yes

Optimal: Picks the shallowest solution. Optimal of path costs are identical.

Chapter 7

Finding the shortest path on weigthed graphs using Dijkstra and priority queues

7.1 Search algorithms with different costs

7.1.1 Uniform cost search

Modify BFS to prioritise cost not depth. expand node with lowest path cost. could be "deep".

This is the same as Dijkstra's algorithm.

Can do this in algo by using heaps

Informed: No

Time: $O(b^?)$

Space: $O(b^?)$

Complete: Yes

Optimal: Yes

Chapter 8

Using heuristics for greedy search and A* search

8.1 Search algorithms with heuristics

8.1.1 Greedy search

Find absolute distance from goal for each node. choose node with shortest distance.

Cost of each is $f(n) = h(n)$, where $h(n)$ is the heuristic cost of node n.

8.1.2 A* search

If the heuristic is admissible, then a* is optimal. Intuitively because the the heuristic steers away from any suboptimal solutions.

Admissible? For all nodes n , h(n)¡=h*(n). where h* is true cost

$f(n) = g(n) + h(n)$

$g(n)$ is the cost to reach n from the current position.

Informed: Yes

Time: Exponential

Space: Big, all nodes kept in memory

Complete: Yes

Optimal: Yes, if the heuristic is admissible

8.1.3 Iterative deepening A*

8.1.4 Identifying heuristics

Generating heuristics for search. we can losen restriction to get a much simpler problem and rank moves by how good they are for loosened problem.

eg for for path to goal, assume can go directly from next place in a straight line.

Chapter 9

The travelling salesman problem

9.1 Travelling salesman problem

9.1.1 Travelling salesman problem

Chapter 10

Constraint Satisfaction Problem (CSP) and Sudoku

10.1 Introduction

10.1.1 Constraint Satisfaction Problem

A CSP problem is one where we don't care about the path, we just want to identify the goal state.

For example, solving a sudoku

10.1.2 Defining a CSP

A CSP has:

- Variables X_i.
- Domain for each variable D_i.
- Constraints C_j.

In a CSP there are a range of variables each with a domain. There are on top constraints on combinations of values.

A solution does not violate any constraint.

To solve, start with no allocations of variables. successor function assigns a value to an unassigned variable. goal test

Use heuristic minimum remaining value MRV: choose variable with fewest remaining legal values

Least constraining value: choose item in domain which constrains the least other moves

Forward checking. keep track of remaining legal moves for each variable. terminate if none left

After each move, update legal moves for each

Implement all this with recursive backtrack function, which returns a solution or failure. This is a depth first search

10.1.3 Arc-consistency

X-Y is arc consistent if all of domain of X is consistnet with some value of Y.

10.1.4 Node-consistency

X is node consistent if all of domain satisfies all unary constraint.

10.1.5 Path-consistency

Arc consistency for additional variables.

10.1.6 Constraint propagation

Constraint propagation can be used to prevent bad choices

We can check for:

- node-consistency
- arc-consistency
- path-consistency

10.1.7 AC-3

AC-3 algorihm makes a CSP arc-consistent

Take all arcs.

It may be possible to break the problem down into sub problems, making the problems much easier to solve.

Can do this before/after other algorithm.

10.1.8 Constraint Satisfaction Problem

Introduction

For active learning, only need to update covariance matrix? just needed to select one with highest variance

Active learning is greedy algorithm to reduce entropy

As we get more info, our posterior becomes our new prior

If we can pick observations to use to update model, can use those with biggest variance

Can be useful if getting y is expensive. requires experiement etc

4 steps:

- Form $p(y_0|x_0, y, X)$ for all unmeasured x_0.
- Choose x_0 with the largest σ_0^2 and observe y_0
- Updated the parameters with this.
- repeat

$\sigma_0^2 = \sigma^2 + x_0^T \Sigma x_0$

Updating Sigma and mu for bayesian linear:

$\Sigma = (\lambda I + \sigma^{-2}(x_0 x_o^T + \sum_{i=1}^n x_i x_i^T))^{-1}$

$\mu = (\lambda\sigma^2 I + x_0 x_0^T + \sum_{i=1}^n x_i x_i^T)^{-1}(x_0 y_0 + \sum_{i=1}^n x_i y_i)$

Once we have an x_0 we can easily get μ_0 by calculating $x_0^T \mu$. Multiplying by the mean weights.

We can also get the variance of the estimate:

$\sigma_0^2 = \sigma^2 + x_0^T \Sigma$

Part IV

Dynamic programming

Chapter 11

Memoisation using search trees for associative arrays

Chapter 12

Using hash tables to create associative arrays

Chapter 13

Dynamic programming and the Bellman equations

13.1 Dynamic programming

13.1.1 Dynamic programming

Dynamic programming is similar to divide and conquer algorithms, in that both solve sub-problems.

However, if dynamic problems, the sub-problems overlap.

13.1.2 Hamilton-Jacobi-Bellman equation

13.1.3 Policies

A policy maps the state onto the action

$a_t = \pi(s_t)$

The policy does not need to change over time, as discounting is constant. That is, if the policy should be different in future, it should be different now.

The policy affects the transition model, and so we have P_π.

Optimal policy

There exists a policy that is better than any other policy, under any starting state.

There is no closed form solution to finding the optimal policy.

There are instead iterative methods.

13.1.4 Bellman equations

We breakdown the value function into an immediate reward, and the discounted value function of the next state.

This is because the expectation function is linear.

$v_\pi(s) = R_{s,\pi(s)} + \gamma \sum_{s'} P_{s,\pi(s)}(s')v_\pi(s')$

We can write this in matrix form.

$v_p i(s) = r_\pi + \gamma P_\pi v_\pi(s)$

We can then solve this:

$v_p i(s) = (I - \gamma P_\pi)^{-1})r_\pi$

This depends on the starting state.

Part V

Lisp

Chapter 14

Functions and recursion

14.1 Functions

14.1.1 Intro

Also called subroutines

14.1.2 Functions

We can call a function: $y = f(x)$

Here x are the local variables to be used.

However the global state may also affect the outcome, so we have:

$y = f(x, z)$

As a result, calling the same function twice with the same inputs can have different outputs.

Side effects. If a function modifies the state outside of its local variables it has side effects.

14.2 Recursion

14.2.1 Stack overflows

Write too many instructions to stack. can be caused by infinite recursion. eg fun x() return x()).

14.3 Other

14.3.1 Global and local variables

Chapter 15

The Church-Turing thesis

15.1 Storing knowledge

15.1.1 Introduction

15.2 The Church-Turing thesis

15.2.1 Introduction

Chapter 16

Imperative and functional programming

16.1 Introduction

16.1.1 Imperative programming

In imperative programming, we say exactly what we want to happen. Each step changes the state.

16.1.2 Functional programming

With functional programming, we write functions to be called. These functions should not depend on the state, outside of local variables.

16.1.3 Side effects

If we remove side effects from all functions then functional programming has no global variables which could affect the output.

Chapter 17

Lisp