Data structures and algorithms

Adam Boult (www.bou.lt)

April 29, 2024

Contents

Part I

Alternatives to arrays: Linked lists and dynamic arrays

Chapter 1

Linked lists

1.1 Single linked lists

1.1.1 Single linked lists

A linked list is a collection of nodes. Each with a value and a pointer to the next element in the list.

If the element is the last in the linked list, the pointer is a null pointer.

The first item is the "head".

1.1.2 Traversing the linked list

Start at the head, and then get the pointer, go to that location and continue.

1.1.3 Advantages and disadvantages of linked lists over arrays

Advantages of linked lists include: + Can insert into the list without moving other elements, just changing the pointers. + Can add to the end of a list without needed the next physical space to be available.

Disadvantages of linked lists include: + Takes time ($O(n)$) to traverse. + Pointers take up memory.

1.2 Write operations on arrays

1.2.1 Inserting to and removing from linked lists

An element can be inserted into a linked list by changing the pointer prior to the element to the element, and setting the pointer for the new element to what

would have previously been the next element.

An element can be removed by setting the pointer for the prior element to the element after the one being removed. Note that this means the "deleted" data still exists, it is just not available in the list.

1.3 Operations involving multiple lists

1.3.1 Slicing linked lists

1.3.2 Filtering linked lists

1.3.3 Merging sorted linked lists

1.3.4 Concatenating linked lists

Chapter 2

Doubly linked lists

2.1 Double linked lists

2.1.1 Doubly linked lists

successor, predecesor and key, null if nec

Chapter 3

Reversing linked lists

3.1 Introduction

3.1.1 Introduction

Chapter 4

Insertion sort on linked lists

4.1 Insertion sort

4.1.1 Insertion sort on arrays

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

Chapter 5

Merging sorted linked lists

5.1 Introduction

5.1.1 Introduction

Chapter 6

Dynamic arrays

6.1 Introduction

6.1.1 Introduction

overallocate on creation. can add more elements in the space.

if go over limit, double (etc) size and move elsewhere

alternative to linked list, some benefits and drawbacks (easier to lookup, harder to insert in middle)

Part II

Stacks, queues and dequeues

Chapter 7

Implementing stacks with arrays or linked lists

7.1 Introduction

7.1.1 Introduction

Chapter 8

Implementing queues with doubly linked lists

8.1 Introduction

8.1.1 Introduction

Doubly linked lists mean you can access the start and end.

A singly linked list which also includes a pointer to the last element in the head could also be used.

Chapter 9

Implementing dequeues with doubly linked lists

9.1 Introduction

9.1.1 Introduction

Part III

Implementing associative arrays

Chapter 10

Binary search trees and sets

10.1 Binary search trees

10.1.1 Introduction

10.2 Using binary search trees to implement sets

10.2.1 Introduction

Chapter 11

Self-balancing binary search trees, including red-black trees

11.1 Introduction

11.1.1 Introduction

Chapter 12

Self-balancing non-binary search trees, including B-trees

12.1 Introduction

12.1.1 Introduction

Chapter 13

Using hash tables to create associative arrays

13.1 Introduction

13.1.1 Introduction

13.1.2 Hash functions

map from key to array offset

13.1.3 Hash collisions

13.1.4 Resolving hash collisions with separate chaining

rather than return offset for array, returns pointer for linked list. items in linked list contain key, so if not correct one can go to next in list

13.1.5 Resolving hash collisions with open addressing

Store the key in the bucket. If the key doesn't match the bucket keep going down until you find it. This can also be used to insert.

13.1.6 Load factor of hash tables

Number of entries over number of possible entries.

Performance deteriorates as load factor increases. can be rehashed with more possible entries.

Chapter 14

Tries

14.1 Introduction

14.1.1 Introduction

Tree of possible words, each branch a letter adding. leaves are possible words, can be used intead of hash table as associative array.

Part IV

Implementing priority queues: Heaps

Chapter 15

Implementing priority queues with a heap

15.1 Introduction

15.1.1 Priority queues

Each item has associated priority. want to be able to take highest priority, add others Can be implemented using a heap; or a self balanacing binary tree Operations: insert with priority; pull highest priorty; is emtpy?

15.1.2 Heaps

Parent greater than or equal to children

Root then always has highest number

Binary heap: just 2 children per node

Chapter 16

Heaps and heapsort

Part V

Graphs and using search trees to find routes between two graph nodes

Chapter 17

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

17.1 Depth-first search

17.1.1 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

17.2 Search algorithms

17.2.1 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.

17.2.2 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 18

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

18.1 Breadth-first search

18.1.1 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 19

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

19.1 Search algorithms with different costs

19.1.1 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 20

Using heuristics for greedy search and A* search

20.1 Search algorithms with heuristics

20.1.1 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.

20.1.2 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

20.1.3 Iterative deepening A*

20.1.4 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.

Part VI

More graph problems

Chapter 21

The travelling salesman problem and the Christofides algorithm

21.1 Travelling salesman problem

21.1.1 Travelling salesman problem

21.1.2 The Christofides algorithm

Returns no more than 50

Chapter 22

Hamiltonian paths and Hamiltonian cycles

22.1 Introduction

22.1.1 Introduction

hamilton problem: Visit each vertex once.

Believed to be intractible (NP).

Chapter 23

Identifying Eulerian paths

23.1 Introduction

23.1.1 Introduction

is there a path which traverses each edge once?

The requirements are:

+ The graph must be connected + At most 2 nodes with odd connections

Part VII

Other

Chapter 24

Constraint Satisfaction Problem (CSP) and Sudoku

24.1 Introduction

24.1.1 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

24.1.2 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

24.1.3 Arc-consistency

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

24.1.4 Node-consistency

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

24.1.5 Path-consistency

Arc consistency for additional variables.

24.1.6 Constraint propagation

Constraint propagation can be used to prevent bad choices

We can check for:

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

24.1.7 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.

24.1.8 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 VIII

Divide and conquer algorithms

Chapter 25

Divide and conquer algorithms and merge sort and quick sort

Part IX

Dynamic programming

Chapter 26

Memoisation using search trees for associative arrays

Chapter 27

Dynamic programming and the Bellman equations

27.1 Dynamic programming

27.1.1 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.

27.1.2 Hamilton-Jacobi-Bellman equation

27.1.3 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.

27.1.4 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.