

Data Structures and Algorithms

More Algorithm Design: Brute Force, Backtracking, Branch&Bound and Dynamic Programming

See references in Goodrich & Tamassia to
Brute Force & Dynamic Programming

1

Brute Force

Key idea: Systematically evaluate many, or all, candidate solutions, often using substantial computational resources.

Good if

- The number of candidate solutions is 'small': **must** be finite.
- Solutions must be identified or ordered
- Solutions can be quickly evaluated

Example: try all possible keys to decrypt a simple cipher e.g. all possible columnar transpositions up to message-length/2.

Often used with other techniques, e.g. to solve 'small' problems in D&C.

2

The drawback is that the solutions to many problems don't have the properties above, e.g. NP hard problems.

Backtracking and Branch&Bound enable more effective exploration of such large solution spaces.

3

Backtracking

Key idea: First organise the candidate solutions to make searching easy, e.g. a graph or tree. Then search the solution space depth-first, and if the node at the end of the current search proves infeasible, mark it as dead and backtrack to the previous node.

Live nodes are part of the current solution under investigation (often on a path).

Dead nodes are part of solutions that have proved to be infeasible.

More example backtracking algorithms in Weiss Section 10.5

4

Maze Search Problem

Find a path through a maze from top left (1,1) to bottom right (4,4) avoiding obstacles

Maze is represented as a matrix:

	1	2	3	4
1	0	0	0	0
2	0	1	1	1
3	0	0	1	0
4	0	0	0	0

5

Solution space is an undirected graph:

1,1	----	2,1	----	3,1	----	4,1
1,2	----	2,2	----	3,2	----	4,2
1,3	----	2,3	----	3,3	----	4,3
1,4	----	2,4	----	3,4	----	4,4

Every path from (1,1) to (4,4) is a solution, but some cross obstacles, and are not feasible.

Search Objective: to find a feasible path from top-left to bottom-right.

6

A Backtracking Alg: Maze search

DFS tries to move Right, Down, Left, Up

Depth First Search

0	0	0	0
0	1	1	1
0	0	1	0
0	0	0	0

Backtrack!

0	0	0	0
0	1	1	1
0	0	1	0
0	0	0	0

7

Depth First Search

0	0	0	0
0	1	1	1
0	0	1	0
0	0	0	0

Path:

Note: Backtracking is easily implemented in Logic languages like Prolog, and in *lazy* functional languages like Haskell.

Exercise: Trace the algorithm for the following maze.

0	0	0	0
0	1	0	1
0	1	1	1
0	0	0	0

8

Branch&Bound

Key idea: Use a breadth first search of the solution space, but **prune** suboptimal solutions

Good for parallelism: there are many solutions to consider simultaneously

But ... multiple solutions consume resources, including memory

9

Branch&Bound Alg: Maze Search

Breadth First Search

```
0 0 0 0
0 1 1 1
0 0 0 0
0 0 1 0    2 paths
```

```
0 0 0 0
0 1 1 1
0 0 0 0
0 0 1 0    2 paths
```

10

```
0 0 0 0
0 1 1 1
0 0 0 0
0 0 1 0    3 paths
```

Top path is infeasible

Prune: chose shorter, or arbitrarily between equal-length

```
0 0 0 0
0 1 1 1
0 0 0 0
0 0 1 0    2 paths
```

11

```
0 0 0 0
0 1 1 1
0 0 0 0
0 0 1 0    1 paths
```

```
0 0 0 0
0 1 1 1
0 0 0 0
0 0 1 0    1 path
```

Path:

12

Dynamic Programming

Key idea: Store optimal solutions to subproblems, and use them to solve bigger problems

Often used to produce an efficient version of a recursively specified problem

More examples in Weiss Section 10.3.2 onwards

13

Fibonacci Numbers

The fibonacci numbers are a sequence of numbers defined as follows (in SML)

```
fun fib 0 = 0
  | fib 1 = 1
  | fib n = fib(n-1) + fib(n-2)
```

i.e. 0,1,1,2,3,5,8,13,21,34

Fibonacci numbers have many properties, e.g. the sum of the squares of two consecutive fibonacci numbers is also a fibonacci number.

14

Naive Fib Function

A recursive definition is natural:

```
public static int fib(int n)
{
  if (n <= 1)
    return n;
  else
    return fib(n-1) + fib(n-1);
}
```

15

Efficiency of fib

fib(5)

16

Dynamic Programming Fib

Idea: store previous fibonacci numbers

```
public static int fib(int n)
{
    int [] fibs;
    fibs[0] = 0;           // Initialise array
    fibs[1] = 1;
    for (int i = 2; i <= n; i++)
        fibs[i] = fibs[i-1] + fibs[i-2];
    return fibs[n];
}
```

Concluding Thoughts

There are many clever algorithms and data structures out there.

Pick suitable algorithms and data structures for your problem, and try to reuse code.

If you implement your own algorithms and data structures: aim for generic code.

Happy Hacking!