Theory Lecture 8

Algorithm Design Strategies

Learning Objectives

- Discuss some approaches to developing fast algorithms
- Deeper analysis of divide and conquer
- Understand exhaustive search, backtracking and pruning
- Understand the concept of a heuristic

Divide and Conquer

Divide and Conquer

We have discussed divide and conquer a number of times in the module.

General strategy

1. Divide the problem up into b subparts
2. Solve each subpart separately
3. Combine the results

We can apply the strategy to subparts as well, down to a base case which is trivial. This is a recursion.

We gain if we can combine the results efficiently.

Example - MergeSort

MergeSort

MergeSort(l : list)	
if l.length==1 then return l	1
m=l.length/2	1
l_1=l(0…m)	$n/2$
l_2=l(m+1…end)	$n/2$
MergeSort(l_1)	$f(n/2)$
MergeSort(l_2)	$f(n/2)$
return Merge(l_1,l_2)	n

$$f(n) = 2f\left(\frac{n}{2}\right) + 2n + 2$$

We divide into two parts, which are sorted, then combine into the result

In lecture 3, we made some simplifications to analyse the complexity as $O(n \log n)$. What about other situations?

Divide and Conquer Complexity

Typically, the form of the complexity of a divide-and-conquer method is

$$f(n) = af\left(\frac{n}{b}\right) + g(n)$$

Subparts computation multiplier

Number of parts

Cost of combination

For example, in MergeSort, $b = 2$, and $a = 2$ because we have to sort both parts. $g(n) = n + 2$, the time to combine the parts.

Master Theorem

Many problems in the form $f(n) = af\left(\frac{n}{b}\right) + g(n)$ can be solved by the master theorem:

1. If $g(n) \in O(n^{\log_b a-\epsilon})$ then $f(n) = \Theta(n^{\log_b a})$
2. If $g(n) \in \Theta(n^{\log_b a})$ then $f(n) = \Theta(n^{\log_b a} \log n)$
3. If $g(n) \in \Omega(n^{\log_b a+\epsilon})$ then $f(n) = \Theta(g(n))$

Here $\epsilon > 0$ is a positive constant.

Example: For MergeSort, $b = 2$, $a = 2$, so we must compare $g(n) = n + 2$ with $n^{\log_2 2} = n$. These have the same growth rate (case 2) so $f(n) = \Theta(n \log n)$ as we found.

Binary Search

Binary Search

```
Search(l : sorted list, x : item)
i=l.length/2
if l(i)==x then return i
if l.length==1 then return None
if l(i)>x then
   return Search(l(0…i),x)
else
   return Search(l(i+1…end),x)
```

Since the list is sorted, we can check the middle item. If it is greater than x, x must be in the first half, otherwise it is in the second.

$$f(n) = f\left(\frac{n}{2}\right) + 3$$

Binary Search Complexity

1. If $g(n) \in O(n^{\log_b a - \epsilon})$ then $f(n) = \Theta(n^{\log_b a})$
2. If $g(n) \in \Theta(n^{\log_b a})$ then $f(n) = \Theta(n^{\log_b a} \log n)$
3. If $g(n) \in \Omega(n^{\log_b a + \epsilon})$ then $f(n) = \Theta(g(n))$

$$f(n) = f\left(\frac{n}{2}\right) + 3$$

For binary search, $b = 2$, $a =$1, so we must compare $g(n) =$3 with $n^{\log_2 1} = 1$. These have the same growth rate (case 2) so $f(n) = \Theta(\log n)$. This allows the search of a sorted list (see lecture 2) in $O(\log n)$.

Big Multiplication

Long multiplication divides the numbers up into digits, and separately multiplies the digits, i.e, 65*23=5*3+5*2*10+6*3*10+6*2*100, the point being that the powers of 10 are done by place shifts. In general, $a = a_0 + a_1 w, b = b_0 + b_1 w$

$$a \cdot b = (a_0 + a_1 w) \cdot (b_0 + b_1 w) = a_0 b_0 + a_0 b_1 w + a_1 b_0 w + a_1 b_1 w^2$$

Assuming the addition of n-digit numbers takes $O(n)$ time and place shift takes $O(1)$, we get

$$f(n) = 4f\left(\frac{n}{2}\right) + O(n)$$

Applying the master theorem, $a = 4, b = 2, n^{\log_b a} = n^2$. Clearly $g(n) \in O(n) \in O(n^2)$ so case 1 applies and $f(n) = \Theta(n^2)$.

This is what we would expect anyway, so this divide-and-conquer shows no benefit.

Search

Search

Consider a problem with the following properties

- A potential solution can be represented by a set of elements. We will can this a PS.
  - For example, in the Hamiltonian path problem the PS is a sequence of vertices in some order
- Each element has k possible values.
- The PS can be checked for accuracy in $O(n^c)$ time, i.e. polynomial time
  - In the Hamiltonian path problem, we can check the PS contains all vertices and has an edge joining each consecutive pair in $O(n)$ where n is the number of vertices

We can solve such a problem by generating each PS and then checking if it is an actual solution

This method is called **exhaustive search** or combinatorial search

Search(2)

Exhaustive search guarantees to find all solutions, but at the expense of high computational cost. Since there are k choices for each element, there are k^n PSs and the time taken is exponential:

$$O(nk^n)$$

For some problems, we can't do much better than this.

Backtracking

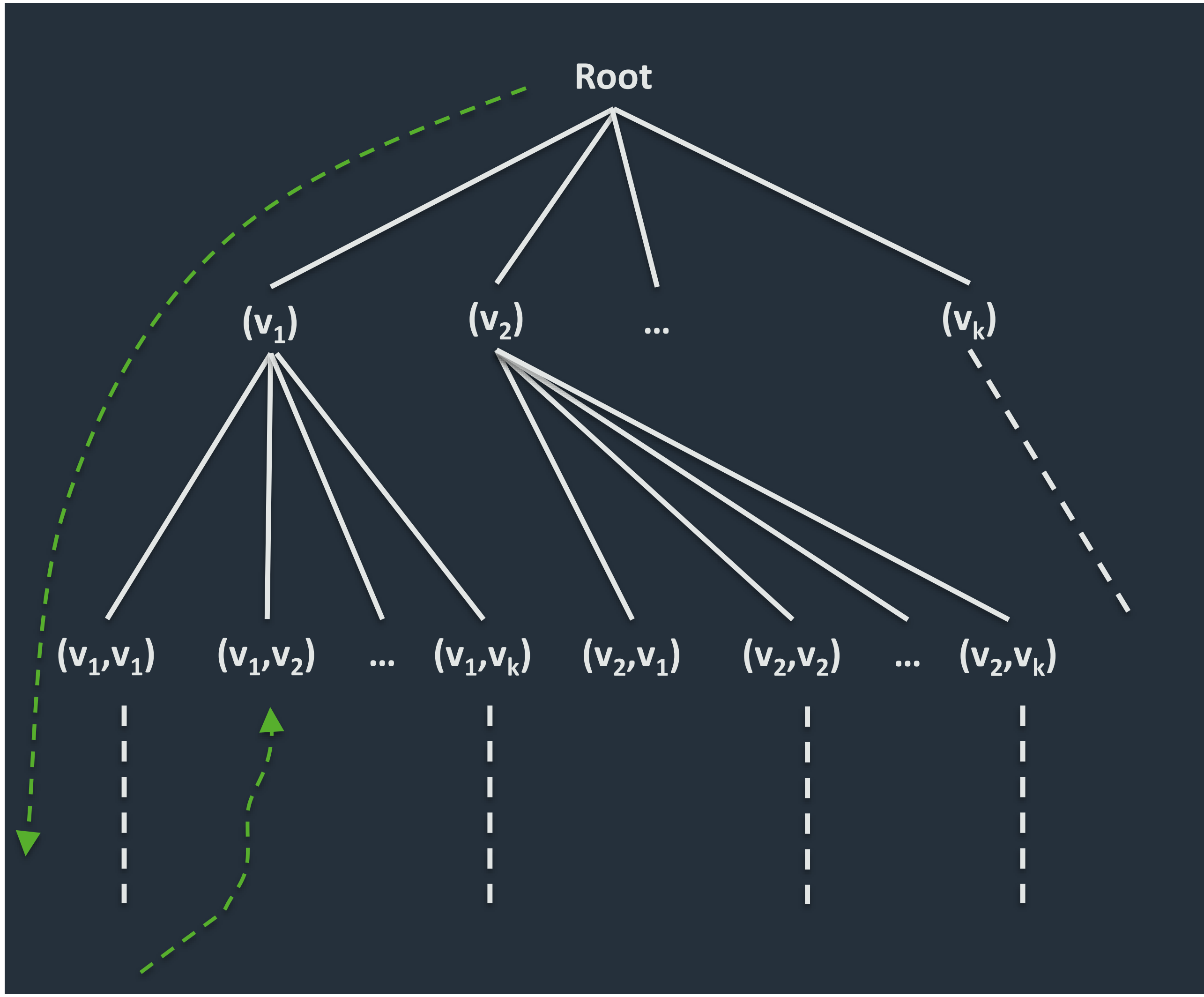
Backtracking is a method of exhaustive search which generates solutions by adding to a partial solution.

The partial potential solution $PPS = (a_0, a_1, \dots, a_{k-1})$, and we extending it by adding all possible extensions $PPS_E = (a_0, a_1, \dots, a_{k-1}, a_k)$

We then repeat the procedure until either

- PPS=PS=a full solution
- We can see than the PPS cannot be part of a solution

This procedure is identical to a DFS of the solution tree on the right, where the solution elements can be $\{v_1, v_2, \dots v_k\}$



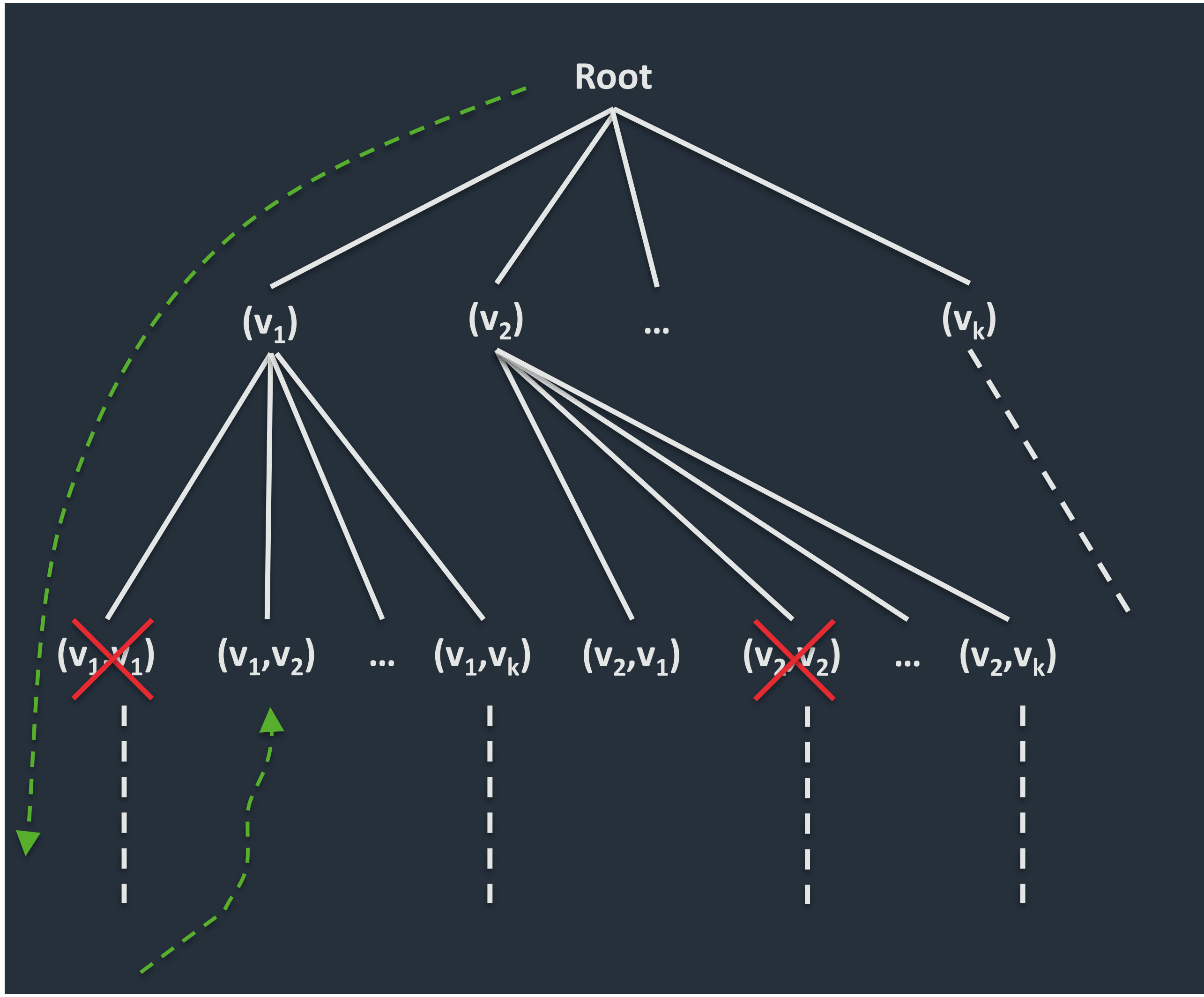
Pruning

We can improve the performance of backtracking if we abandon partial solutions as soon as we know they cannot be a solution.

On the right, imagine that repeated values are not possible in a solution (e.g. Hamiltonian path).

We do not need to consider these branches any further, they are **pruned**.

Example: TSP. We keep track of the shortest route found so far. As soon as a partial route becomes longer than this, we can prune that branch of the search.



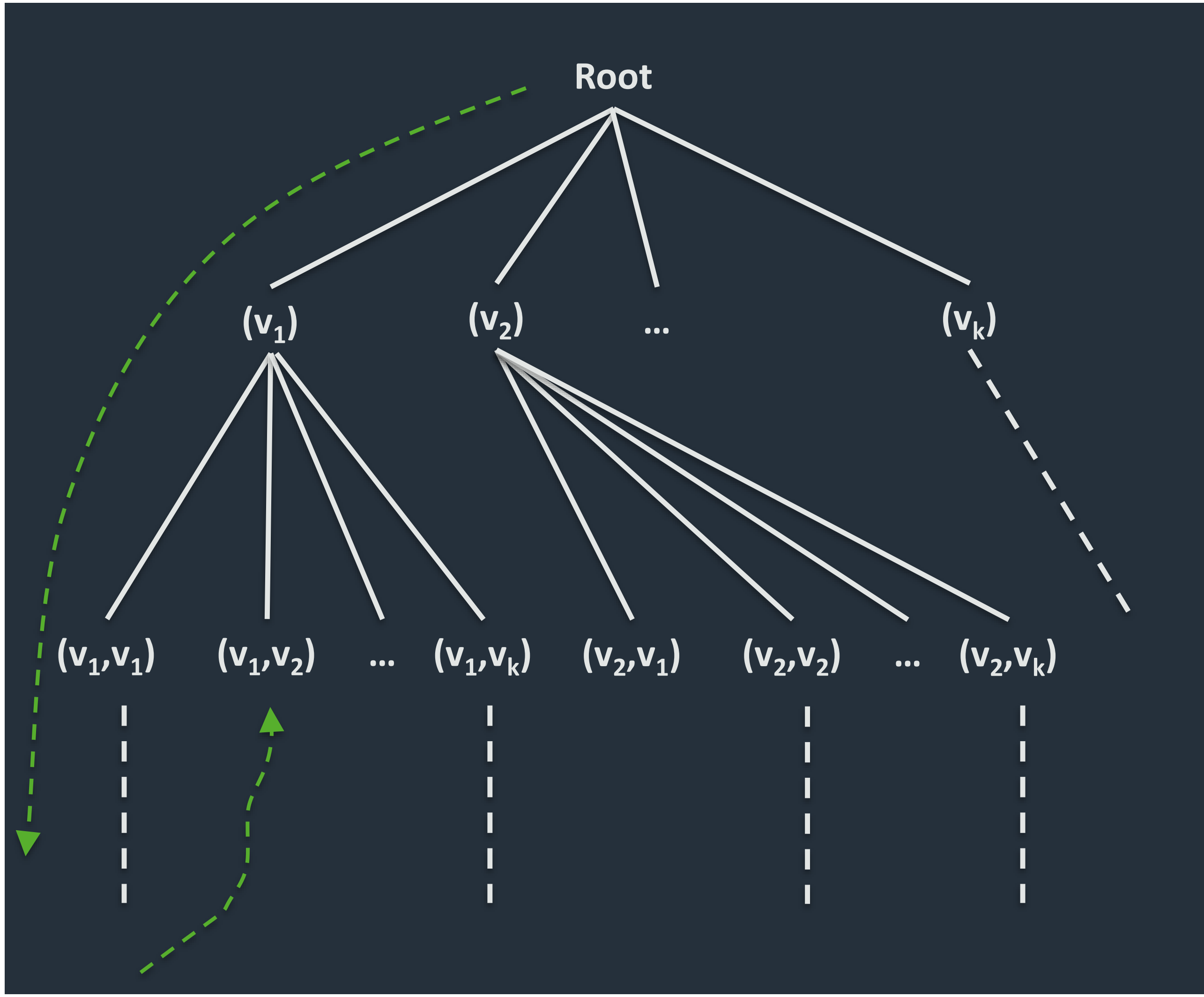
Backtracking Algorithm

Backtracking

```
Backtracking(pps : partial solution)
if IsSolution(pps) then output(pps)
if PruneSolution(pps) then return
foreach epps in Extend(pps)
    Backtracking(epps)
endfor
```

Need to define three things

- Criterion for valid solution
- How to extend a PPS by one element
- When a partial solution can be pruned [optional]



Heuristic

Heuristic

A heuristic is a method or rule that is a shortcut, or a more practical approach to solving a problem than an exact algorithm.

Typically, applying a heuristic leads to an approximate, or non-optimal solution.

Example: TSP

1. Start at a random vertex
2. Take the shortest step to an unvisited vertex
3. Mark the vertex visited
4. End if no unvisited vertices
5. Repeat from 2.

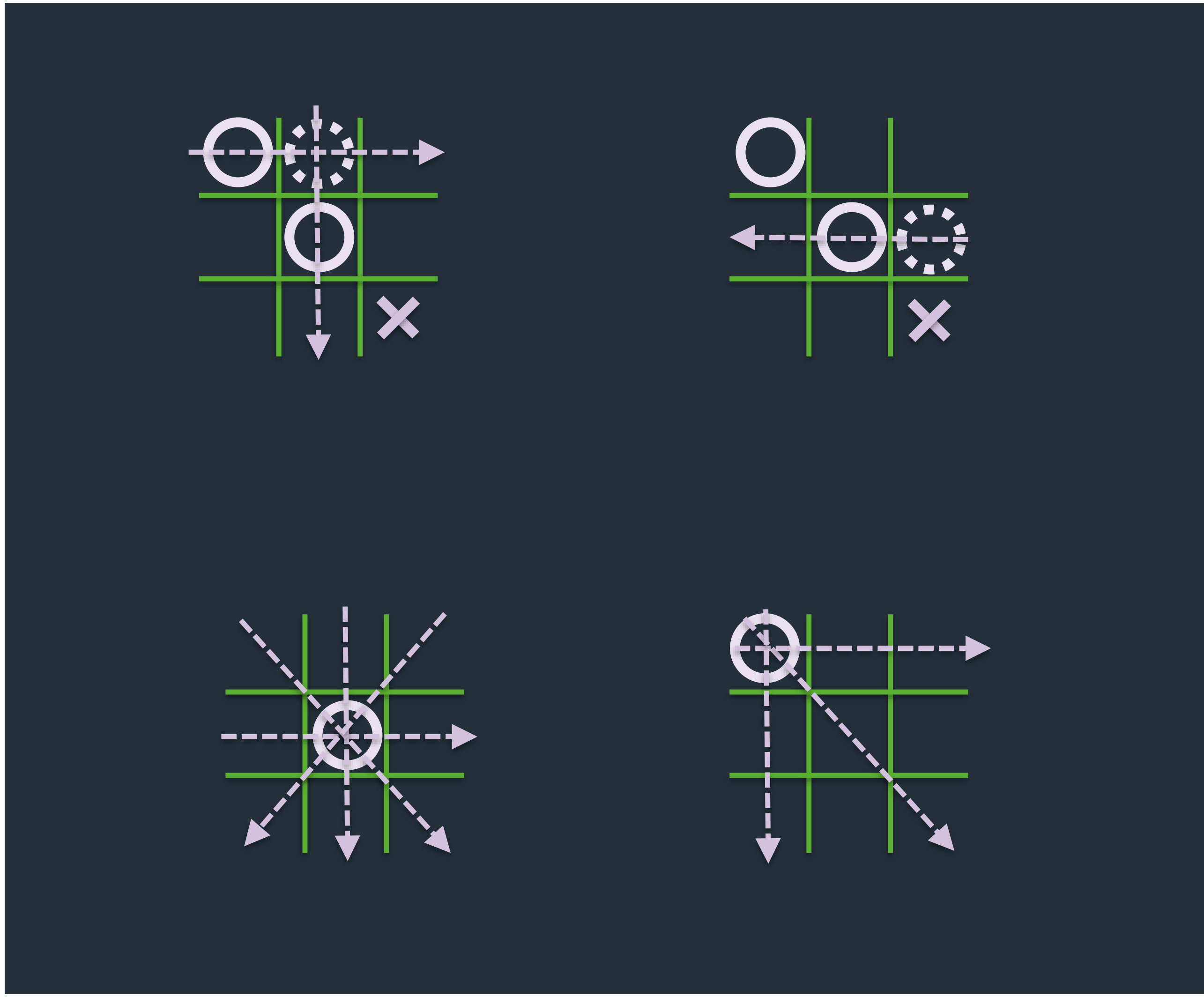
This is non-optimal, the path is on average around 25% longer than the shortest.

Heuristic Example

In the game noughts-and-crosses, there are 255,168 elements in the full space of games.

Rather than searching all games, we can apply a heuristic to favour good moves.

Choose a move which creates the maximum number of chances of three in a row.



Summary

Understand:

- Divide and conquer
- Application of the master theorem
- Exhaustive search
- Backtracking and pruning
- The idea of a heuristic to approximate problems

Read

- Skiena, Chapters 5, 9
- Try exercises 5.4, 5.9, 9.9

Next

- Optimisation algorithms