Goal-directed searching

Dr Yuhua Li
School of Computer Science & Informatics

(Content adapted from slides by Dr Louise Knight)

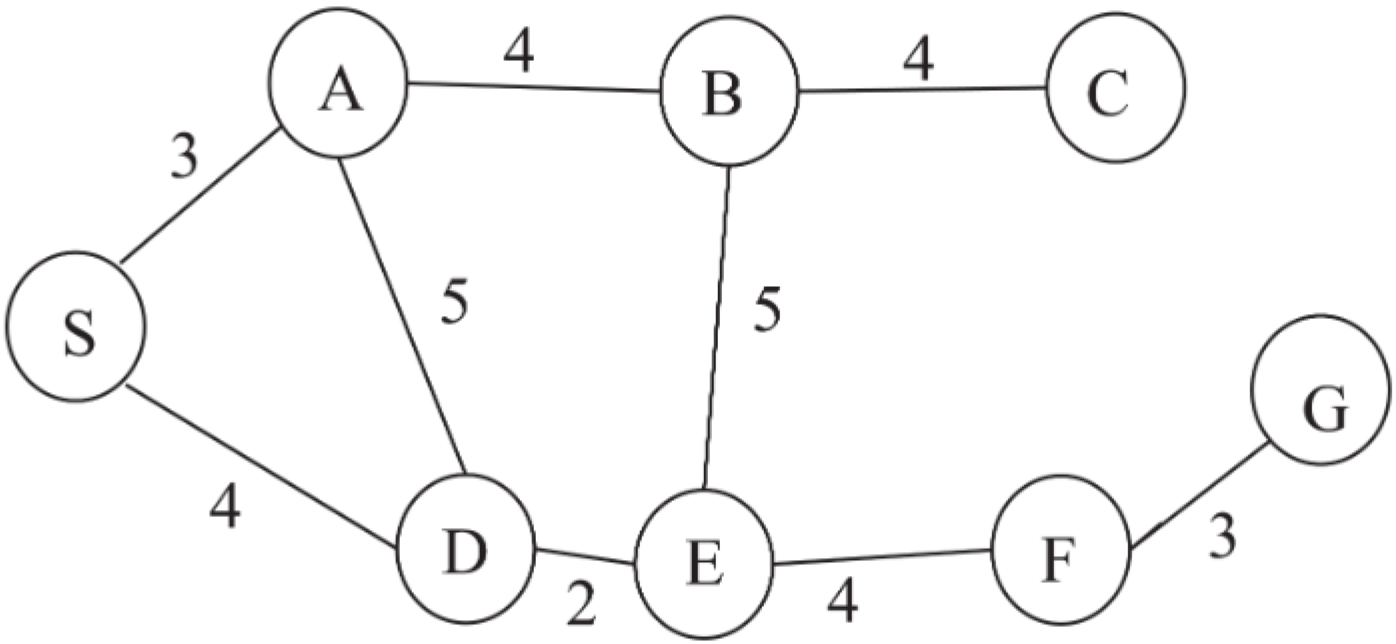
Outline of lecture

- Introduction to searching
- Goal-directed searching
- Breadth-first search and Depth-first search
- Exhaustive search
- Branch-and-bound search
- 0-1 knapsack problem

Introduction to searching

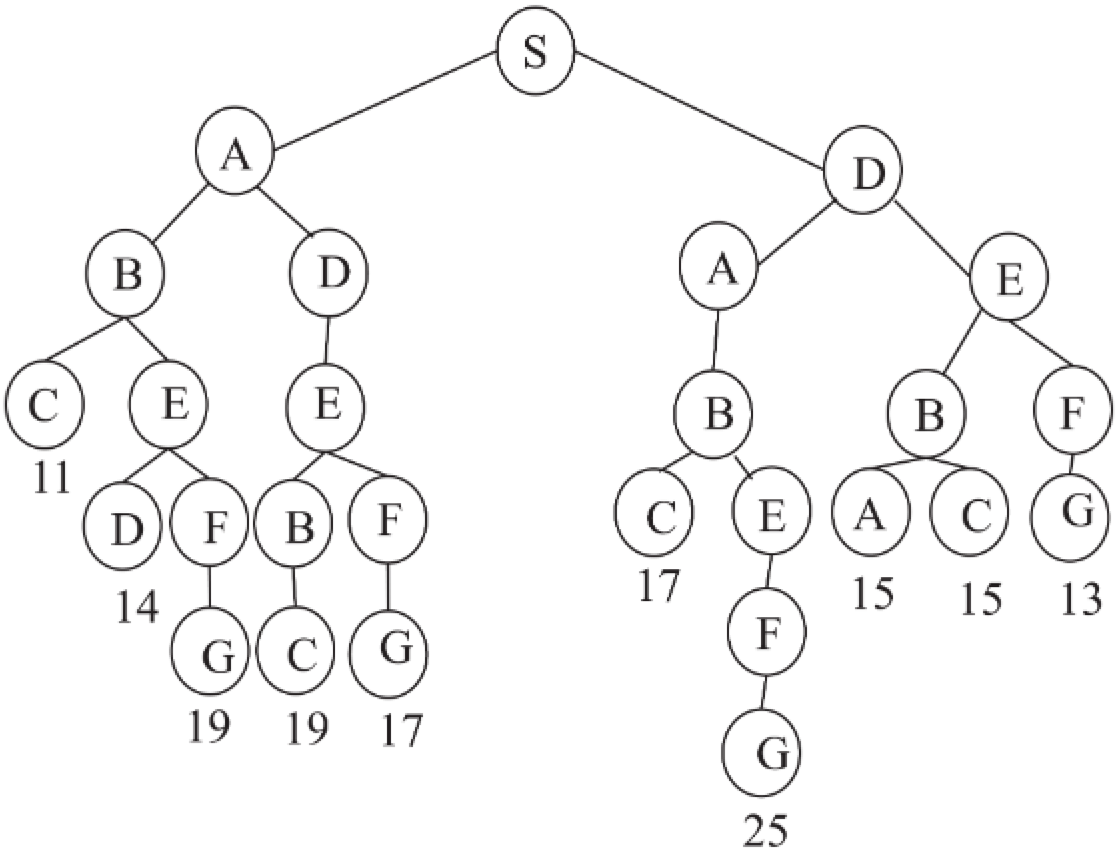
- Search problems pop up everywhere – e.g. exploring alternatives when problem solving, pattern matching for internet/database search
- Two main types of search we will be covering:
  - Goal directed search – exploring alternative routes through the state space of a problem to find a solution or goal
  - Linear search (pattern matching)

A basic search problem



A path is to be found from the start node, *S*, to the goal node, *G*.

A generic search tree



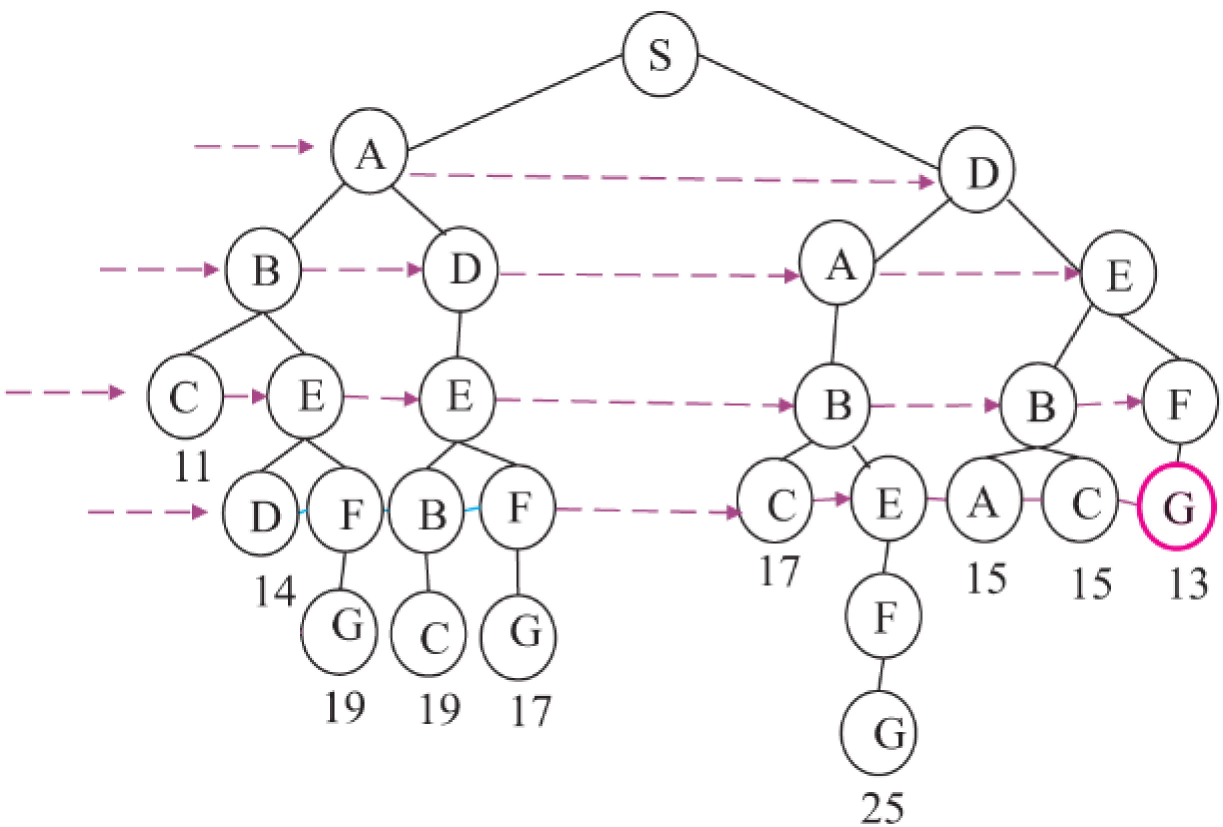
The search tree shows all possible paths from *S* to *G*, with no cycles, i.e. no node is revisited. The numbers are accumulated distances. Each node is a partial path, not just a single location.

Finding paths

- Finding a path involves two kinds of effort:
  - The effort to **find** either some path or the shortest path
  - The effort to actually **traverse** the path
- Things to take into account:
  - Do you want to go from *S* to *G* often? Or just once?
  - Do you need to find the best path, or will any path do?
- Breadth-first search and depth-first search are used to find any path – they differ in the order they traverse the nodes of the tree

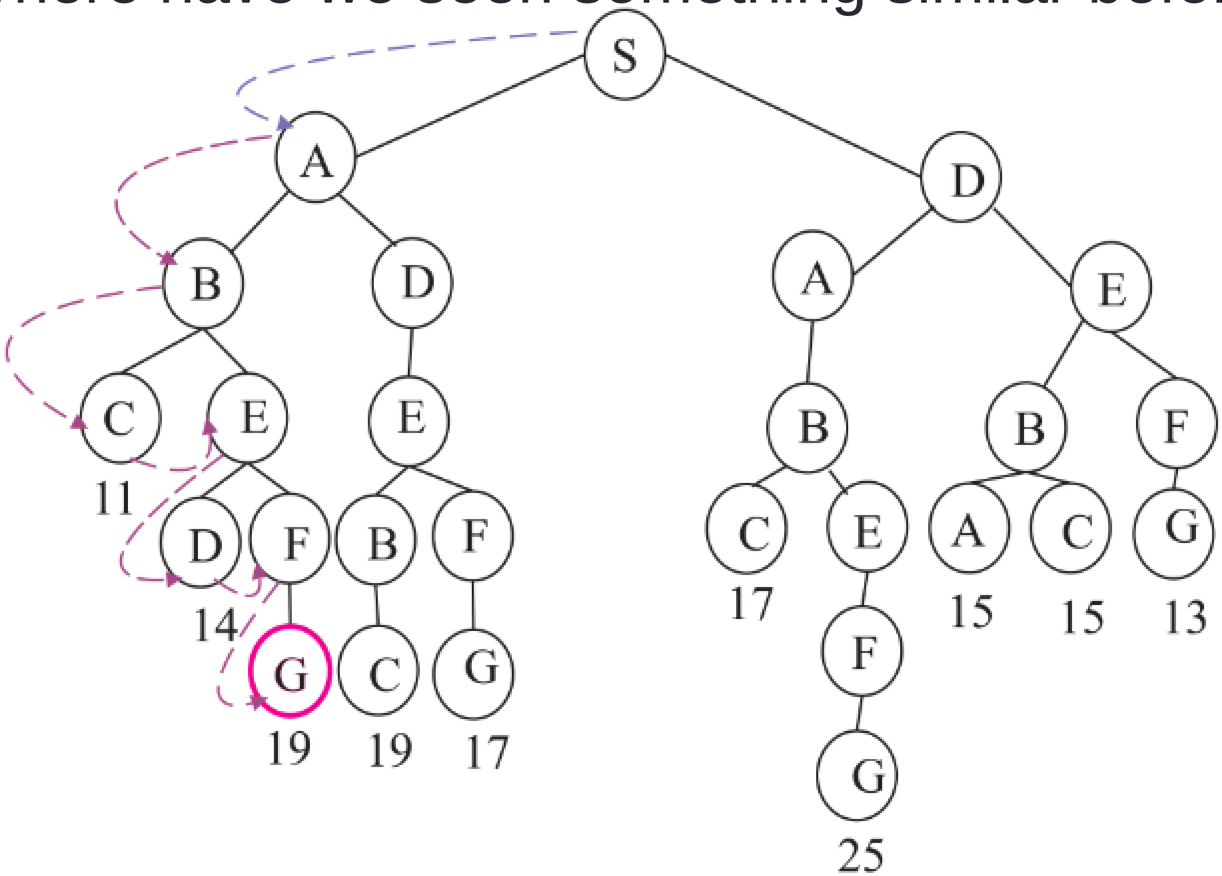
Breadth-first search

Look at the nodes at one level before inspecting the next level down



Depth-first search

Make a headlong dash to the bottom of the tree along the leftmost branches, backtracking when it reaches a dead end. (Where have we seen something similar before?)



BFS versus DFS

- BFS pushes uniformly into the tree
- DFS dives into the tree
- BFS is good when the number of alternatives is not too large
- DFS is good when blind alleys don't get too deep

Exhaustive search

- Exhaustive search explores all possible paths then picks the best
- It can be accomplished by extending either breadth-first or depth-first search
- Instead of stopping when you first encounter the goal, you carry on to the bitter end!
- Exhaustive search may be the only option for some problems
- However, specific knowledge about a problem can help prune the search

Improving the search for the shortest path

- Expand the paths in order of least accumulated cost so far
- Expand the paths on the basis of an underestimate of the remaining distance from a particular node to the goal. Distance remaining can be (under) estimated with a straight line (as the crow flies) measure
- If two or more paths reach a common node, delete all those paths except the one that reaches the common node with the minimum cost

Branch-and-bound search

1. Form a queue of partial paths. Initial queue consists of the zero-length, zero-step path from the root node to nowhere.
2. Until the queue is empty or the goal has been reached, determine if the first path in the queue reaches the goal node.
    1. If the first path reaches the goal node, do nothing.
    2. If the first path does not reach the goal node:
        1. Remove the first path from the front of the queue.
        2. Generate all the path's children by extending one step.
        3. Add the new paths to the queue.
        4. Sort the queue by the sum of the cost accumulated so far and a lower bound estimate of the cost remaining, with the least cost paths in front.
        5. If two or more paths reach a common node, delete all those paths except for the one that reaches the common node with the minimum cost.
3. If the goal node has been found, announce success and print the path; otherwise announce failure.

0-1 knapsack problem

- Thief robbing a shop finds n items
- i^{th} item has profit p_i and weight w_i (these are integers)
- Thief's knapsack has a capacity W
- Want to maximise profit
- Called 0-1 knapsack problem because each item must either be taken or left behind; cannot take a fraction of an item, cannot take an item more than once
- Complexity = $O(2^n)$ if solved by brute strength, as each of the n items can either be included or excluded from the knapsack (will see this in solution tree)

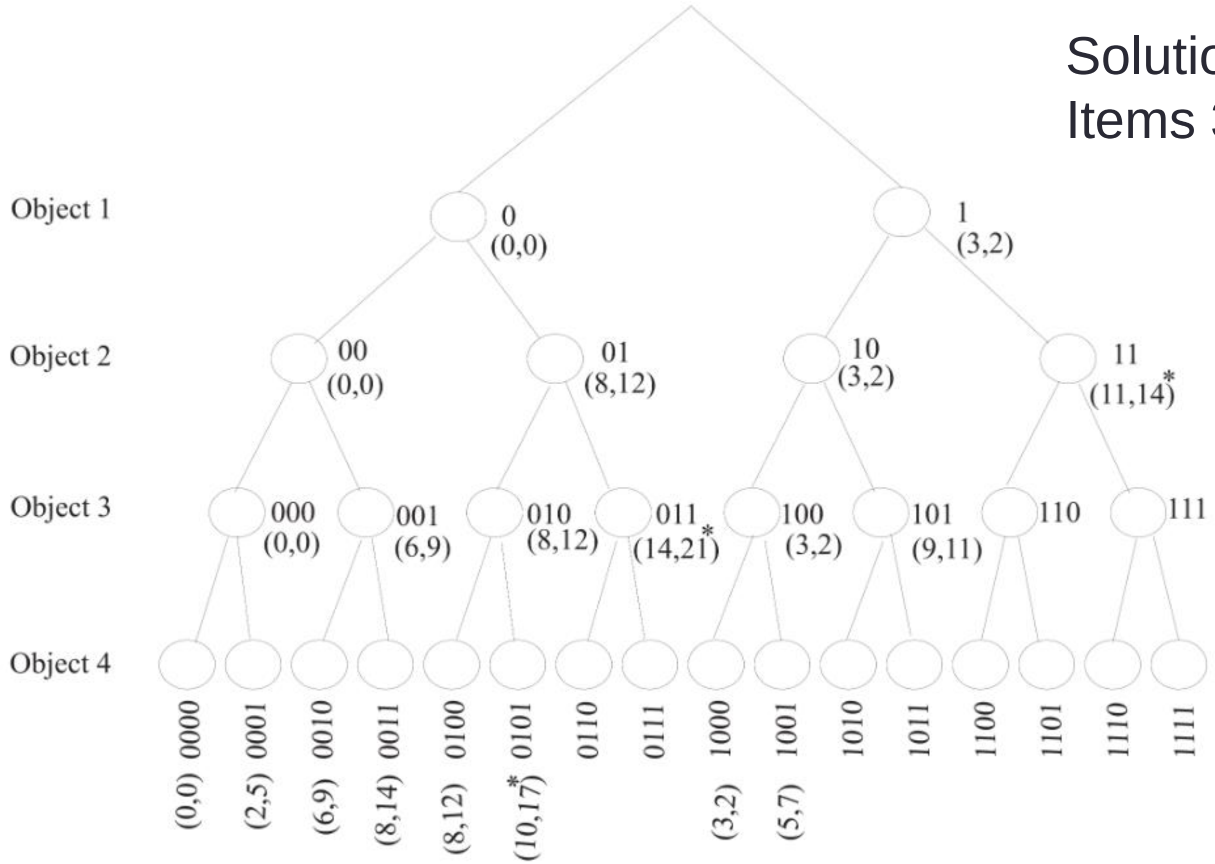
Example

The problem consists of one knapsack with capacity 9, and 4 items with the following weights and profits:

Item no.	Weight	Profit
1	3	2
2	8	12
3	6	9
4	2	5

Solution tree for example

Solution = (8, 14)
Items 3 and 4



0 = object not selected. 1 = object selected.
(*w*,*p*) denotes the total weight and profit of objects selected.
* indicates exceeded capacity.

Towards a branch-and-bound approach

- Sort items in sequence of profit/weight ratio
- Attempt to pack most profitable items per unit weight first
- From this formulation, partial solutions can be extrapolated with **overestimates**
- Overestimates are used in place of the **underestimates** used in the B&B algorithm for shortest path. Why?
- An upper bound can also be easily evaluated

Lower and upper bounds

- Pack the items in sequence of highest to lowest profit to weight ratio
- Until you reach the **break item**
  - Including the break item will exceed the knapsack capacity
  - Excluding the break item will not exceed the constraint
- A lower bound will fill up in sequence, and exclude the break item

A 10 item problem

- Capacity = 165
- Items are already in correct profit/weight sequence

Item no.	Weight	Profit	Profit/weight
1	23	92	4
2	31	57	1.84
3	29	49	1.69
4	44	68	1.55
5	53	60	1.1321
6	38	43	1.1316
7	63	67	1.06
8	85	84	0.988
9	89	87	0.978
10	82	72	0.88

Lower bounds

- Weights: $w_1 + w_2 + w_3 + w_4 = 23 + 31 + 29 + 44 = 127$
- Weights: $w_1 + w_2 + w_3 + w_4 + w_5 = 180$
- Break item is item 5 (knapsack capacity = 165)
- A **lower bound on profit** is sum of profits of first four items = 92 + 57 + 49 + 68 = 266
- Can find a **better lower bound** by ignoring break item and continuing down list for smaller items that may fit. Weights: $w_1 + w_2 + w_3 + w_4 + w_6 = 165$
- **Profit for improved lower bound** is 266 + 43 = 309

Upper bounds

- An **upper bound** can be calculated by filling the knapsack up to, but not including the break item
- Then adding a proportion of the break item to reach the full knapsack capacity
- Finally rounding down the total profit to the nearest lower integer
- Upper bound =

$p_1 + p_2 + p_3 + p_4$ + (capacity – $(w_1 + w_2 + w_3 + w_4)$) x p_5/w_5 =
266 + (165 – 127) x (60/53) = 309.02

- Rounding down gives an **upper bound** of 309, the same as the lower bound

We have a solution!

- Upper bound = (improved) lower bound
- And the solution is best profit = 309, packing items 1, 2, 3, 4, and 6
- We were lucky!
- Solutions are not usually this easy to find
- Comparing the lower and upper bounds is the first stage of our branch and bound algorithm

Exercise for knapsack problem

1. Work out profit/weight ratios, and sequence them
2. Evaluate lower bound
3. Evaluate upper bound

Capacity = 15

Item no.	Weight	Profit	Profit/weight
1	3	9	
2	7	16	
3	2	8	
4	1	1	
5	4	18	
6	3	6	