Backtracking algorithms

Backtracking introduction

- **Backtracking** is a type of algorithm that is a refinement of brute force search methods.
- In backtracking, we check multiple solutions but also find a method by which some can be eliminated without being explicitly examined, by using specific properties of the problem.
- They are normally a depth-first search of the set of possible solutions.
  - During the search, if an alternative doesn't work, the search backtracks to a choice point, which presented different alternatives, and tries the next alternative.
  - When the alternatives at this choice point are exhausted, the search returns to the previous choice point and tries the next alternative there.
  - If there are no more choice points, the search fails.

Backtracking strategy

- Since backtracking is applied to problems in which the sequence of items is chosen from a set where that sequence satisfies some criterion, the choices at each stage can be represented by branching to corresponding nodes of a state tree.
- The backtracking approach then involves a depth-first (or pre-order) search of the state tree for the problem.
- A general recursive outline method, which includes pruning of the state tree where nodes are not promising, is shown on the next slide.

Backtracking strategy

```
checkNode(node v)
if  promising(v) then
      if (there is a solution at v) then
                write the solution
      else
                for (each child u of v)
                          checkNode(u)
                end for
      end if
end if
```

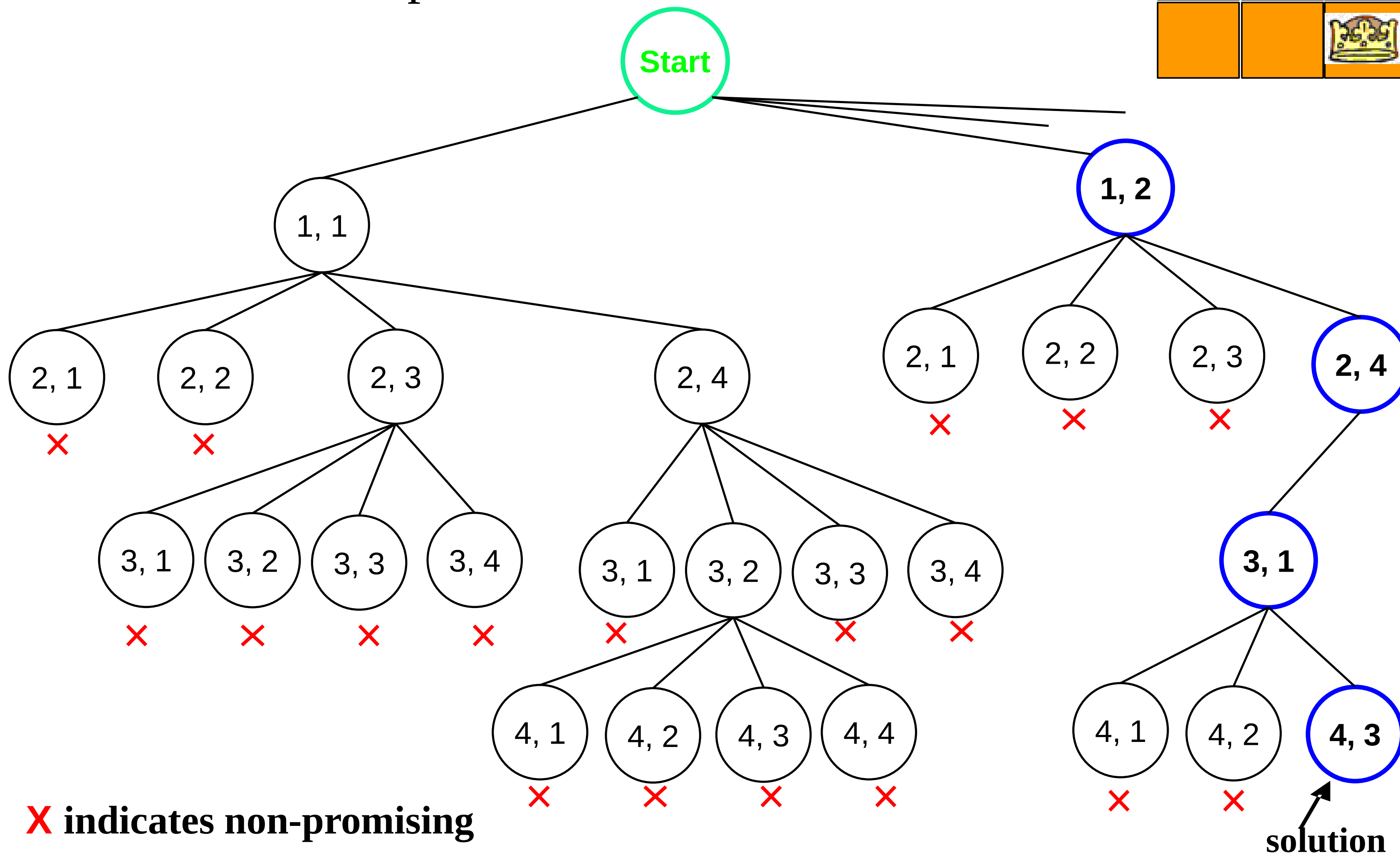
- The definition of **promising** and the **solution condition** depend upon the problem being solved.

N-queens problem

- The n-queens problem is a classical search problem which requires you to place **n** queens on an **n x n** chessboard, so that no two queens can attack each other.
  - No two queens may be in the same row, column, or diagonal

N-queens problem: state tree

- Let's look at that process as a state tree.



State space tree construction for 4-queens

- For n-queens on n x n chessboard, we can immediately simplify matters by realising that no 2 queens can be in the same row.
- Start at level-0 in the tree: the root
- Create the candidate solutions by constructing a tree (called ***state space tree***) in which
  - the column choices for the 1st queen (the queen in row 1) are stored in level-1 nodes in the tree,
  - the column choices for the 2nd queen (the queen in row 2) are stored in level-2 nodes in the tree,
  - …
- A path from the root to a leaf is a candidate solution. In total, there are 4*4*4*4 = 256 candidate solutions.

Backtracking solution

- Use pre-order tree traversal: depth-first search
- Visit a child (left first) of root at (1, 1)
- Check if placing a queen at (1, 1) is promising, as it is the 1st queen placed on the chessboard, so it is promising
  - Visit the child of (1, 1) at (2, 1), check if placing a queen at (2, 1) is promising. It is non-promising as it is in the same column of the 1st queen, so go back (backtrack) to (1, 1)
  - Visit the next unvisited child of (1, 1) at (2, 2), check if placing a queen at (2, 2) is promising. It is non-promising as it is in the diagonal of the 1st queen, so go back to (1, 1)
- Repeat above process until a solution is found or the entire state space tree is traversed.

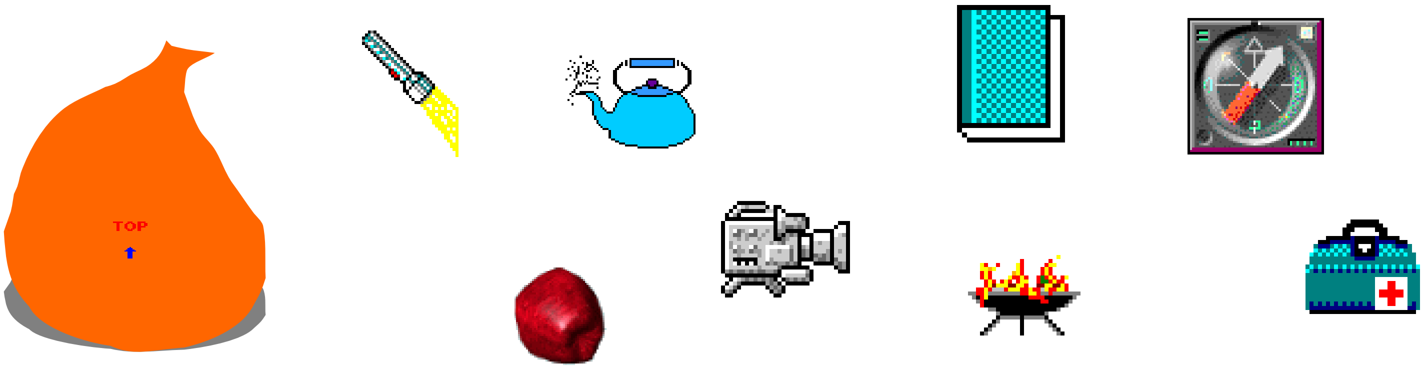
N-queens problem: pseudocode

- The detailed pseudo-code to solve this problem would be:

```
queens(index i)                          //initially called with queens(0)
if promising(i) then
      if  i = n  then                    //solution found
            print out col[1] through col[n]
      else
            for j = 1 to n
                  col[i+1] = j      //set queen in column j
                  queens(i+1)       //check this position
            end for
      end if
end if
```

0/1 knapsack problem reminder

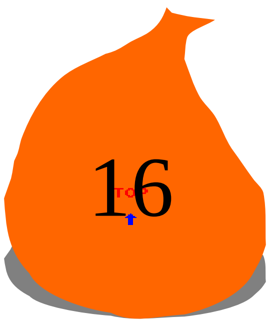
- The 0/1 knapsack problem is a classic optimisation problem.

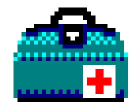


- Hiker wishes to take **n** items on a trip where the weight of item **i** is $\mathbf{w_i}$., profit $\mathbf{p_i}$.
- The items are to be carried in a knapsack whose weight capacity is **M**.
- When sum of item weights > **M**, some items must be left behind.
- Which items should be taken/left to maximise the total profit **P**?

0/1 knapsack with backtracking

- Let's try a new setup using profit density again (p/w ratio):

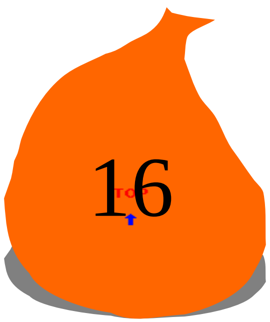


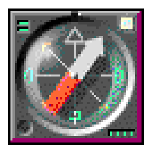
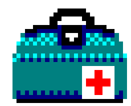
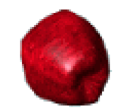
w=2	w=5	w=10	w=5
			
p=40	p=30	p=50	p=10
p/w=20	p/w=6	p/w=5	p/w=2

- The items have been ordered by p/w and will be selected and tested in that order.

Maximum bound on profit with fractional knapsack

- As a method of checking if a route down the binary selection tree is profitable or not we are going to use a criteria called maximum bound on profit.



w=2	w=5	w=10	w=5
			
p=40	p=30	p=50	p=10
p/w=20	p/w=6	p/w=5	p/w=2

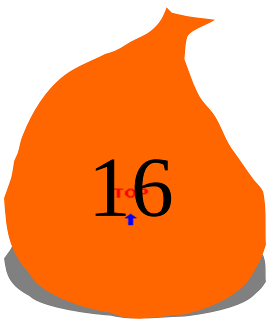
40 + 30 + (12–2–5)*5

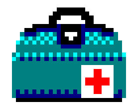
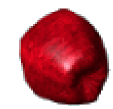
- Taking the items as ordered by p/w.

= a maximum profit bound of 115

Maximum bound on profit with fractional knapsack cont./

- Continuing with the same example but this time omitting the first item:



w=2	w=5	w=10	w=5
			
p=40	p=30	p=50	p=10
p/w=20	p/w=6	p/w=5	p/w=2

30 + 50 + (16-5-10)*2

= a maximum profit bound of 82

- So we can discount this route.

Maximum bound on profit with fractional knapsack algorithm

```
1   start at root of  state tree
2   calculate max bound obtainable on profit up to the weight limit (including
    fractions of an item)
3   if max bound > highest profit seen so far i.e. that node is promising so explore
4    while total weight not exceeded
5           explore down left branches i.e. taking items
6           and, if appropriate, recording actual profit to that leaf node
7      end while
8   end if
9   backtrack to unexplored right branch
10  take right branch (omitting item)
11  repeat from Step 2
```

Fractional example cont./

- We started by calculating the maximum possible profit (which we did on Slide 12) from being able to use all the items.
- Step 2: start our tree with node 1 and put the maximum potential profit in that node.
- Step 3: take item 1 (by adding a left child for node 1) and check whether overweight: weight is 2 kg so within weight limit (of 16).
- Step 4: take item 2 (by adding a left child for node 2) and check whether overweight: weight is 7 kg so within weight limit (of 16).
- Step 5: take item 3 but the weight limit is exceeded so this is not a solution. Hence, put an X in node 4 and write “weight exceeded”.

Fractional example cont./

- Step 6: backtrack to an unexplored right child: this is the child of node 3. We now calculate a new maximum potential profit without choosing item 3. In fact, we can put all the remaining 3 items into the basket without going overweight and this gives us an actual profit of 80.
  - Note that we do not stop at node 5 but continue to the appropriate leaf nodes thus exhausting all possibilities.
  - Node 7 gives another actual profit but its less than 80.
- We now have a possible solution. Let's try to find a better one.

Fractional example cont./

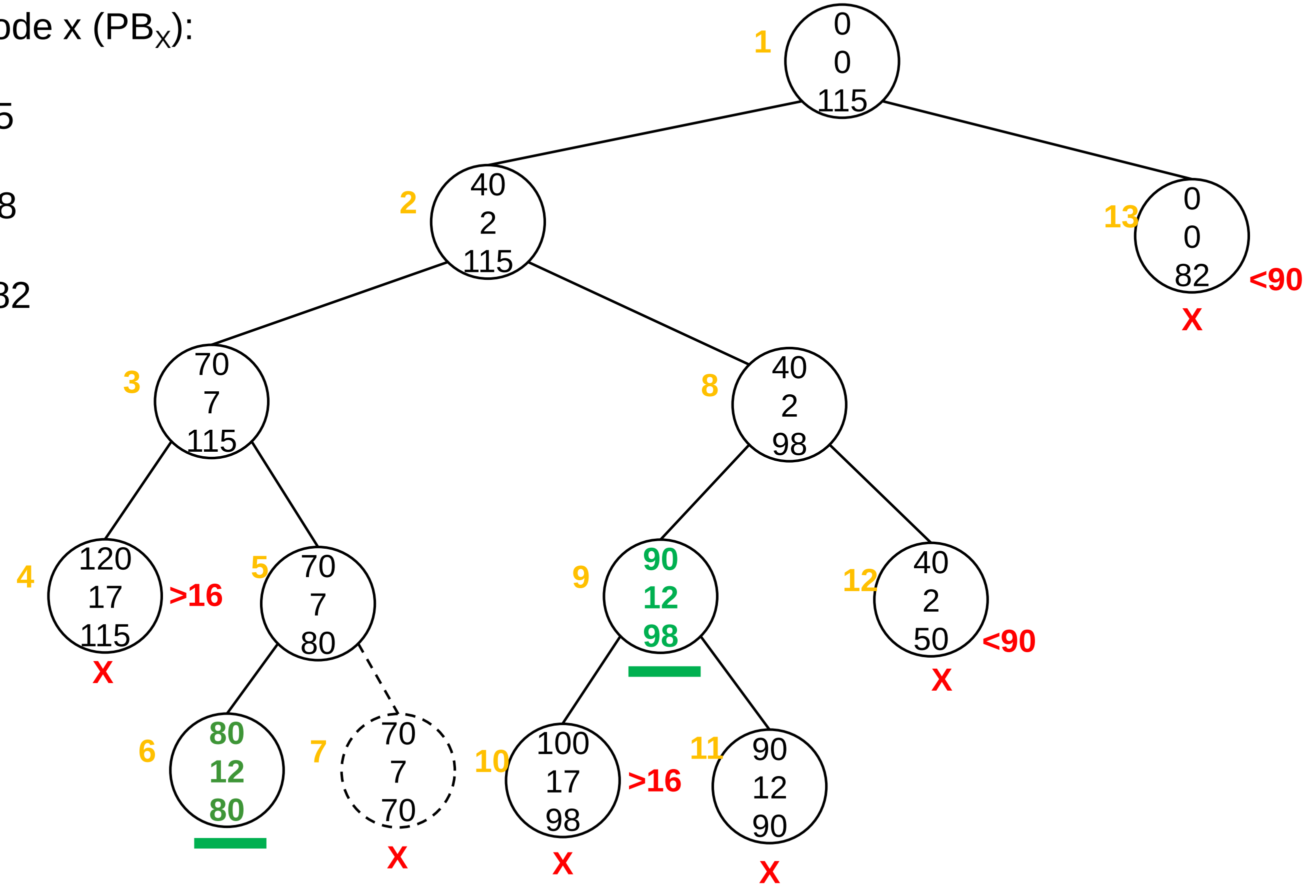
- Step 7: the next unexplored right child is node 2. We now calculate the maximum potential profit by taking item 1, (part of) item 3, (part of) item 4 etc until the weight limit is reached.
  - This maximum profit is item 1, item 3 and 4/5 of item 4 giving a profit of 98.
  - We continue down the tree as 98 is greater than our present best of 80 i.e. this is a promising route which needs exploring.
- Step 8: add node 9 which could still lead to this better profit.
- The route down its left child leads to the weight limit exceeded (node 10) and the route down its right child confirms an actual profit of 90 – the best so far.

Fractional example cont./

- We now have a better possible solution. Let's try to find an even better one.
- Step 9: backtracks again – this time to the right child of node 8 but leaving out items 2 & 3 leads to a lower potential profit: so we write “< best profit so far” underneath node 12 and go no further down that path.
- Step 10: backtracks similarly to node 1 but leaving out node 1 is not a promising solution so that route is not pursued either.
- We have now completed our algorithm for the whole tree and know how to achieve that best profit of 90.

Calculate profit bound (PB) at node x (PB_X):

$PB_1 = 40+30+(16-2-5)*5 = 115$
$PB_5 = 40+30+10 = 80$
$PB_8 = 40+50+(16-2-10)*2 = 98$
$PB_{12} = 40+10 = 50$
$PB_{13} = 30+50+(16-5-10) *2= 82$



Numbers in each node are in order as follows: