Backtracking

Stefan D. Bruda

CS 317, Fall 2025

When dynamic programming does not work

- We use backtracking
  - Commonly used to make a sequence of decisions to build a recursively defined solution satisfying given constraints
  - In each recursive call we make exactly one decision which is consistent with all the previous decisions
  - Example:

algorithm RecKnapsack(i, C, n, p, w)**:** (handle the i-th object)
  if $i > n$ **then return** $(0, \langle\rangle)$
  else
    $(p_-, X_-) \leftarrow$ RecKnapsack$(i+1, C, n, p, w)$ (do not pick item i)
    if $w_i \leq C$ **then**
      $(p_+, X_+) \leftarrow$ RecKnapsack$(i+1, C-w_i, n, p, w)$ (pick item i if we can)
    else
      $(p_+, X_+) \leftarrow (0, \langle\rangle)$
    return MaxFst$(\{(p_-, \langle 0\rangle + X_-), (p_+ + w_i, \langle 1\rangle + X_+)\})$

- Alternative to backtracking: brute force
  - Generate all possible complete sequences of decisions one by one and check if they yield a solution
  - Backtracking has a chance of doing better since it stops when a sequence is hopeless
  - Example: Generate all n-digits in lexicographic order, check that each such a number yields the optimal 0/1 Knapsack solution

n-QUEENS

- Given an $n \times n$ chess board, and n queens, place each i-th queen on the i-th row so that no two queens check each other
  - Intermediate result: $\langle x_1, x_2, \ldots, x_i \rangle$, $i \leq n$
  - Constraints: x_j and x_k, $j \neq k$ are neither the same nor on the same diagonal
  - Decision: placement of one more queen
- Brute force: generate and then check all the possible sequences $\langle x_1, x_2, \ldots, x_n \rangle \rightarrow \Theta(n^{n+1})$ time
- Backtracking:

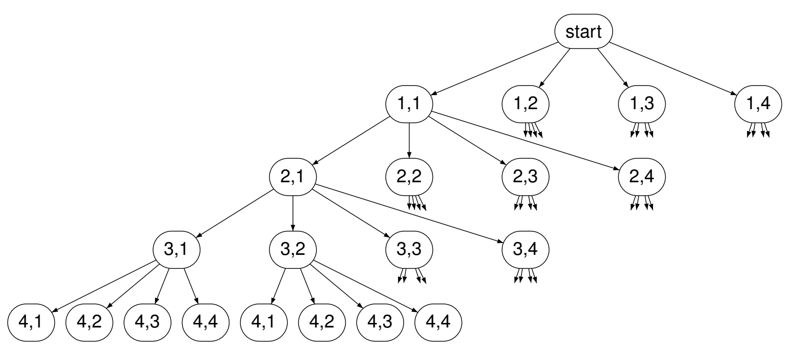
algorithm QUEENS($\langle x_1, x_2, \ldots, x_i \rangle$):
    if $i = n$ **then return** $\langle x_1, x_2, \ldots, x_n \rangle$
    else
        for $j = 1$ **to** n **do**
            if PROMISING($\langle x_1, x_2, \ldots, x_i, j \rangle$) **then**
                QUEENS($\langle x_1, x_2, \ldots, x_i, j \rangle$)

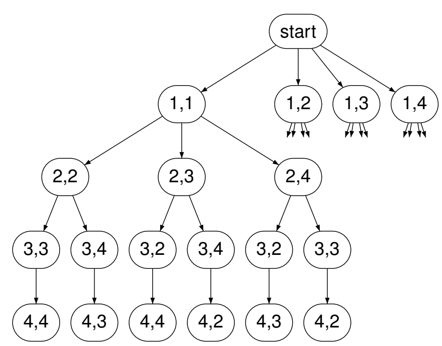
algorithm PROMISING($\langle x_1, x_2, \ldots, x_i \rangle$):
    $k \leftarrow 1$
    safe $\leftarrow$ TRUE
    while $k < i \wedge$ *safe* **do**
        if $x_i = x_k \vee |x_i - x_k| = i - k$ **then**
            safe $\leftarrow$ FALSE
        $k \leftarrow k + 1$
    return *safe*

- Common patterns:
  - Traverse tree of states (aka state space)
    - Different decisions yield different next states
  - Carry over enough information between recursive calls to check feasibility

n-QUEENS (CONT'D)

- Whole state space ($n = 4$): $4^4 = 256$ leaves and $1 + 4 + 4^2 + 4^3 + 4^4 = 341$ nodes
  - Slight optimization of the state space: no two queens can be on the same column ($1 + 4 + 4 \times 3 + 4 \times 3 \times 2 + 4 \times 3 \times 2 \times 1 = 65$ nodes)
  - Backtracking expands only 61 nodes





- For $n = 8$ we have 19,173,961 nodes overall, 109,601 optimized, and 15,721 expanded by backtracking

Optimal BST

$$A_{i,j} = \begin{cases} p_i \text{ (root } i\text{)} & \text{if } i = j \\ \min_{i \le k \le j}(A_{i,k-1} + A_{k+1,j} + \sum_{m=i}^{j} p_m) \text{ (root } k\text{)} & \text{if } i < j \end{cases}$$

- Brute force: generate all possible trees, retain the optimal one
- Backtracking for the optimal cost:

algorithm $\text{CostBST}(i,j)$:
- **if** $i = j$ **then return** p_i
- **else if** $i > j$ **then return** 0
- **else**
  - $m \leftarrow \infty$
  - **for** $k = i$ **to** j **do**
    - $b \leftarrow \text{CostBST}(i, k-1)$
    - $c \leftarrow \text{CostBST}(k+1, j)$
    - $a \leftarrow b + c + \sum_{m=i}^{j} p_m$
    - **if** $a < m$ **then**
      - $m \leftarrow a$
  - **return** m

- Backtracking for the optimal BST:

algorithm $\text{OptBST}(i,j)$:
- **if** $i = j$ **then return** $(p_i, \text{Node}(i))$
- **else if** $i > j$ **then return** $(0, \text{Null})$
- **else**
  - $m \leftarrow (\infty, \text{Null})$
  - **for** $k = i$ **to** j **do**
    - $(b, l) \leftarrow \text{OptBST}(i, k-1)$
    - $(c, r) \leftarrow \text{OptBST}(k+1, j)$
    - $a \leftarrow b + c + \sum_{m=i}^{j} p_m$
    - **if** $a < m$ **then**
      - $m \leftarrow (a, \text{Node}(k, l, r))$
  - **return** m

- When we solve a problem using backtracking we effectively solve a whole family of related problems

Traveling salesman

- Brute force: try all the permutations, retain the one with minimal cost
- Backtracking: With $g(i,S)$ the length of the shortest path starting at i and going through all the vertices in S back to 1,

$$g(i,S) = \begin{cases} \min_{(i,j)\in E}(w(i,j)) & \text{if } S = \emptyset \\ \min_{j \in S}(w((i,j)) + g(j, S \setminus \{j\})) & \text{otherwise} \end{cases}$$

algorithm $\text{TS}(i,S)$:
- **if** $S = \emptyset$ **then**
  - **return** $\min_{(i,j)\in E}(w(i,j))$
- **else**
  - $m \leftarrow \infty$
  - **forall** $j \in S$ **do**
    - $a \leftarrow w((i,j)) + \text{TS}(j, S \setminus \{j\})$
    - **if** $a < m$ **then**
      - $m \leftarrow a$
  - **return** m

algorithm $\text{TSX}(i,S)$:
- **if** $S = \emptyset$ **then**
  - **return** $(\min_{(i,j)\in E}(w(i,j)), \langle i,j \rangle)$
- **else**
  - $(m,k) \leftarrow (\infty, 0)$
  - **forall** $j \in S$ **do**
    - $(a,b) \leftarrow \text{TSX}(j, S \setminus \{j\})$
    - $a \leftarrow a + w((i,j))$
    - **if** $a < m$ **then**
      - $(m,k) \leftarrow (a,b)$
  - **return** $(m, \langle i \rangle + k)$

GENERAL FORM OF BACKTRACKING

algorithm GENERICBKT(v):
  if v is a solution **then**
    Return solution
  else
    foreach child u of v **do**
      if PROMISING(u) **then**
        GENERICBKT(u)

Effectively implements a depth-first traversal of the state space of the given problem

- Possibly pruning the state space using PROMISING
- Improvement over the brute force
- However, the call to PROMISING may be missing for some problems
  - In this case backtracking offers no advantage run time-wise over brute force

GRAPH COLORABILITY

- The graph m-colorability problem: Given an undirected graph G and an integer m, can the vertices of G be coloured with at most m colours such that no two adjacent vertices have the same colour
  - The smallest possible m is called the chromatic number of G
    - The maximum chromatic number of a planar graph is 4

algorithm COLOURS($\langle c_1, \ldots, c_i \rangle, G = (V, E)$):
  if $i = n$ **then return** $\langle c_1, \ldots, c_n \rangle$
  else
    for $c = 1$ **to** m **do**
      if PROMISING($\langle c_1, \ldots, c_i, c \rangle$) **then**
        COLOURS($\langle c_1, \ldots, c_i, c \rangle$)

algorithm PROMISING($\langle c_1, \ldots, c_i \rangle$):
  $j \leftarrow 1$
  safe $\leftarrow$ TRUE
  while $j < i \wedge$ *safe* **do**
    if $(i, j) \in E \wedge c_i = c_j$ **then**
      safe $\leftarrow$ FALSE
    $j \leftarrow j + 1$
  return *safe*

Better backtracking for optimization problems

- In optimization problems we can keep track of the best solution found so far and avoid expanding nodes if they would lead to a worse solution:

algorithm GenericBktOpt(v):
- **if** v is a solution **then** Return solution
- **else if** Value(v) is better than *bestsofar* **then** *bestsofar* $\leftarrow$ Value(v)
- **else if** Promising(v) **then**
  - **foreach** child u of v **do** GenericBktOpt(u)

  - Value(v) is an upper/lower bound for all the solutions below v
  - *bestsofar* is a global variable maintained between different branches
  - Promising must reject nodes of less value than *bestsofar*
- Case in point: 0/1 Knapsack revisited
  - The state space is binary (left child = pick item, right child = do not pick item)
  - Each state stores three values
    1. accumulated profit
    2. accomolated weight
    3. the upper bound Value() = the profit that can be made if the problem was fractional Knapsack
  - A node is not promising if either
    - The accumulated weight is larger than the capacity C, or
    - The upper bound is less than the maximum profit made so far

0/1 Knapsack revisited

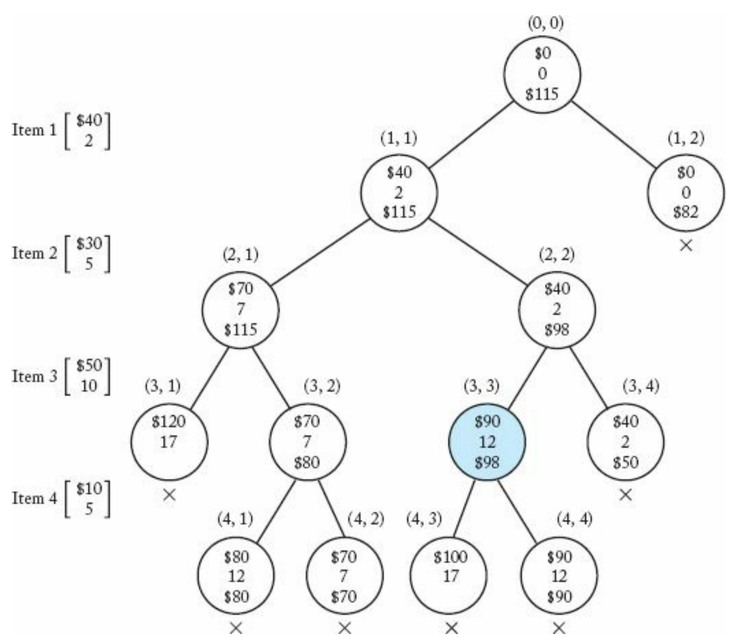
algorithm Knapsack():
- *bestsofar* $\leftarrow 0$
- **for** $i = 1$ **to** n **do** $result_i \leftarrow$ False
- KnapsackRec(0, 0, 0)
- **return** (*bestsofar*, *bestset*)

algorithm KnapsackRec(i, *profit*, *weight*):
- **if** $weight \leq C \wedge profit > bestsofar$ **then**
  - *bestsofar* $\leftarrow$ *profit*
  - *bestset* $\leftarrow$ *result*
- **if** Promising(i) **then**
  - $result_{i+1} \leftarrow$ True
  - KnapsackRec($i+1$, $profit + p_{i+1}$, $weight + w_{i+1}$)
  - $result_{i+1} \leftarrow$ False
  - KnapsackRec($i+1$, *profit*, *weight*)

algorithm Promising(i):
- **if** $weight \geq C$ **then** **return** False
- **else**
  - $j \leftarrow i + 1$
  - $bound \leftarrow profit$
  - $W \leftarrow weight$
  - **while** $j \leq n \wedge W + w_j \leq C$ **do**
    - $W \leftarrow W + w_j$
    - $bound \leftarrow bound + p_j$
    - $j \leftarrow j + 1$
  - **if** $k \leq n$ **then**
    - $bound \leftarrow bound + (C - W) \times p_j / w_j$
  - **return** $bound > profit$

Obj:	1	2	3	4
p	40	30	50	10
w	2	5	10	5
p/w	20	6	5	2

$C = 16$



BRANCH & BOUND

- Similar to backtracking, but only for optimization problems
- Every time a state is considered its “value” is compared with the best solution candidate obtained so far
- Also changes the order of evaluation from depth first to
  - Breadth-first branch & bound
  - Best-first branch & bound where each node is associated a bound that denotes how “good” that node is

algorithm BRANCH&BOUND(v, *bestsofar*):

```
open ← ⟨⟩
EnQueue(v, open)
bestsofar ← Value(v)
while open ≠ ⟨⟩ do
    u ← DeQueue(open)
    foreach child u of v do
        if Value(u) > bestsofar then
            bestsofar ← Value(u)
        if Bound(u) > bestsofar then
            EnQueue(u, open)
```

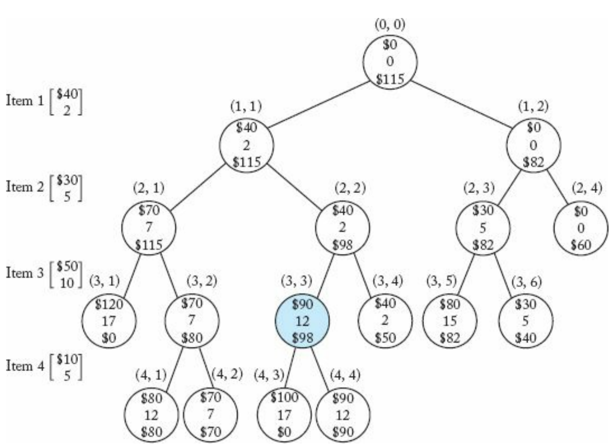
- *open* = queue → breadth-first branch & bound
- *open* = priority queue with key VALUE → best-first branch & bound

BRANCH & BOUND (CONT’D)

Obj:	1	2	3	4
p	40	30	50	10
w	2	5	10	5
p/w	20	6	5	2

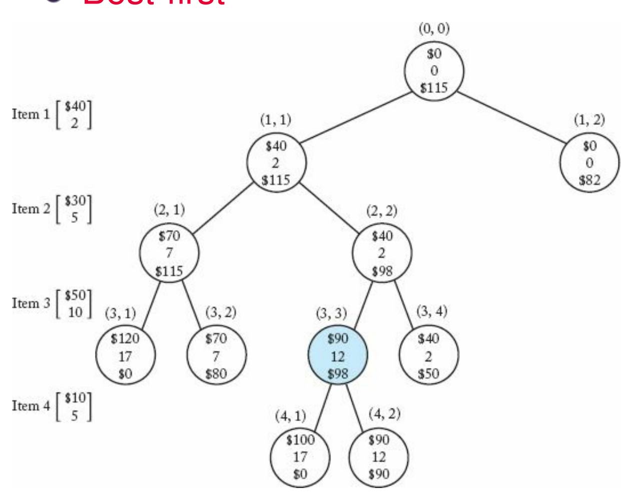
$C = 16$

- Breadth-first



When it is time to enqueue (2,3) its bound (82) is larger than *bestsofar* (70) so we enqueue and later expand. However, by the time we dequeue it *bestsofar* has already changed to 98.

- Best-first



Substantially smaller tree than in the case of breadth-first branch & bound.