Backtracking

Stefan D. Bruda

CS 317, Fall 2024

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_i 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:

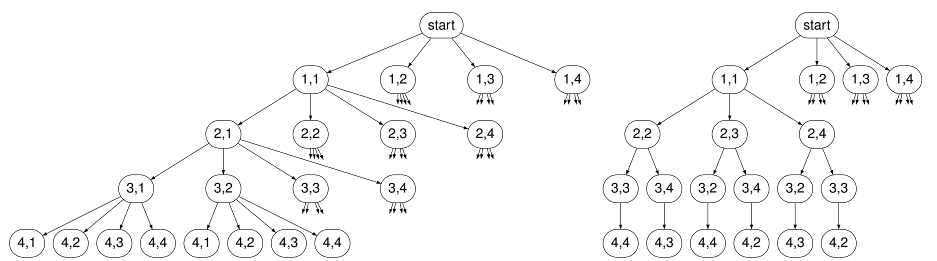
algorithm Queens($\langle x_1, x_2, \ldots, x_i\rangle$):
 if $i = n$ **then return** $\langle x_1, x_2, \ldots, x_n\rangle$
 else
  for $j = 1$ **to** n **do**
   if Promising($\langle x_1, x_2, \ldots, x_i, j\rangle$) **then**
    Queens($\langle x_1, x_2, \ldots, x_i, j\rangle$)

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 CostBST(i, j):
  if i = j then return p_i
  else if i > j then return 0
  else
    m ← ∞
    for k = i to j do
      b ← CostBST(i, k − 1)
      c ← CostBST(k + 1, j)
      a ← b + c + Σ_{m=i}^{j} p_m
      if a < m then
        m ← a
    return m
```

- Backtracking for the optimal BST:

```
algorithm OptBST(i, j):
  if i = j then return (p_i, Node(i))
  else if i > j then return (0, Null)
  else
    m ← (∞, Null)
    for k = i to j do
      (b, l) ← OptBST(i, k − 1)
      (c, r) ← OptBST(k + 1, j)
      a ← b + c + Σ_{m=i}^{j} p_m
      if a < m then
        m ← (a, 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 TS(i, S):
  if S = ∅ then
    return min_{(i,j)∈E}(w(i, j))
  else
    m ← ∞
    forall j ∈ S do
      a ← w((i, j)) + TS(j, S \ {j})
      if a < m then
        m ← a
    return m
```

```
algorithm TSX(i, S):
  if S = ∅ then
    return (min_{(i,j)∈E}(w(i, j)), j)
  else
    (m, k) ← (∞, 0)
    forall j ∈ S do
      (a, b) ← w((i, j)) + TSX(j, S \ {j})
      if a < m then
        (m, k) ← (a, b)
    return (m, 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(⟨c_1, ..., c_i⟩, G = (V, E)):
  if i = n then return ⟨c_1, ..., c_n⟩
  else
    for c = 1 to m do
      if Promising(⟨c_1, ..., c_i, c⟩) then
        Colours(⟨c_1, ..., c_i, c⟩)
```

```
algorithm Promising(⟨c_1, ..., c_i⟩):
  j ← 1
  safe ← True
  while j < i ∧ safe do
    if (i, j) ∈ E ∧ c_i = c_j then
      safe ← False
    j ← 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 ← 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

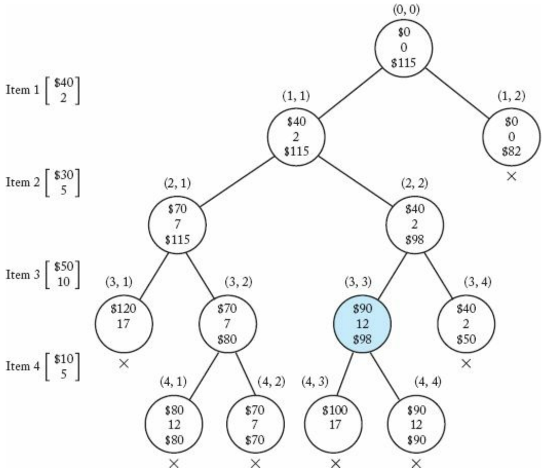
```
algorithm Knapsack():
    bestsofar ← 0
    for i = 1 to n do result_i ← False
    KnapsackRec(0, 0, 0)
    return (bestsofar, bestset)

algorithm KnapsackRec(i, profit, weight):
    if weight ≤ C ∧ profit > bestsofar then
        bestsofar ← profit
        bestset ← result
    if Promising(i) then
        result_{i+1} ← True
        KnapsackRec(i+1, profit+p_{i+1}, weight+w_{i+1})
        result_{i+1} ← False
        KnapsackRec(i + 1, profit, weight)

algorithm Promising(i):
    if weight ≥ C then return False
    else
        j ← i + 1
        bound ← profit
        W ← weight
        while j ≤ n ∧ W + w_j ≤ C do
            W ← W + w_j
            bound ← bound + p_j
            j ← j + 1
        if k ≤ n then
            bound ← bound + (C − W) × 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)
```

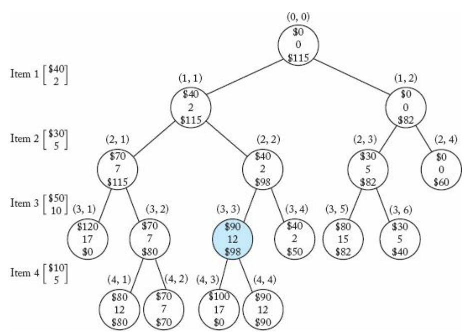
- *open* = queue → breadth-first branch & bound
- *open* = priority queue with key Value → best-first branch & bound

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

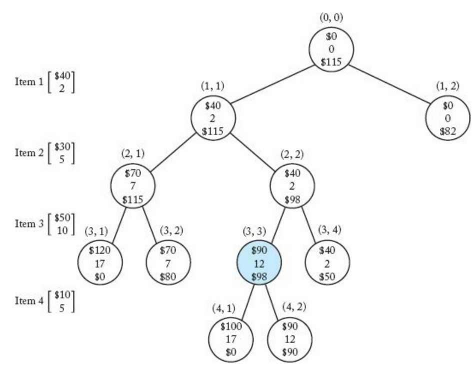
$C = 16$

- 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.