Generating New Subproblems

An appropriate framing of alternatives and a clever representation may also be effective.

- e.g. 0-1 knapsack
  - subproblem requires the set of items remaining to consider
    - if the items left to be considered are all at one end of an array, a single index k + the original collection of items is sufficient to define S for the subproblem
  - the choice and alternatives
    - take or not take the next item (process input)
      - S is defined by k+1 for the subproblems
    - which item to take next (produce output)
      - combine with considering items in some order → S is defined by k+1, k+2, k+3, … for the subproblems

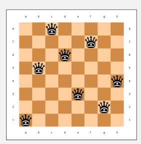
Key Points – Making Backtracking Practical

- recursive backtracking is generally not practical without additional effort
  - DFS is O(n+m) where n = O(b^h)
    - b = branching factor – number of next options for each choice
    - h = length of longest path – (maximum) number of choices made to get to a complete solution

The Effectiveness of Reducing Search

Cleverness in reducing the searching done is essential for there to be any hope of an efficient solution.

The 8 queens problem has 178,462,987,637,760 possible placements.



Tactics.

- only 16,777,216 placements with one queen per column
- only 15,720 placements when conflicts with previously-placed queens are considered
- only half as many placements when symmetry is exploited – only consider n/2 possible rows for the first column placement

Key Points – Making Backtracking Practical

- strategies

effective strategies tend to be highly problem-specific

  - *reduce the size of the search space* – formulate the problem to reduce b, h
    - prefer a choice with fewer possible values
      - e.g. process input "take or not take" instead of produce output "which to take next" for choose-a-subset problems
    - exploit problem characteristics to reduce the number of next possibilities without skipping possible solutions
      - e.g. *n* queens – there will be exactly one queen per column, so "place the next queen" really means choosing a row for the queen in the next column instead of choosing any of the remaining spaces
    - impose an ordering on the series of decisions to limit redundant paths
      - e.g. *n* queens – place queens in columns from left to right
      - e.g. 0-1 knapsack – for "choose next item to take", order the items in some way and only look forward for the next choices
    - if the output size is smaller than the input size, prefer produce output to process input

Key Points – Making Backtracking Practical

- strategies

effective strategies tend to be highly problem-specific

  - *reduce the amount of the search space explored* – prune branches that don't contain the (only) solution
    - don't pursue dead ends – only consider legal next possibilities (prune invalid partial solutions)
      - e.g. *n* queens – only place a queen if it doesn't attack already-placed queens
      - e.g. 0-1 knapsack – only take an item that fits in the pack
    - don't pursue solutions that are already bad (for optimization problems)
      - prune if the partial solution is worse that the best-so-far complete solution
      - prune if this branch can't lead to a better solution – *branch and bound*
    - prune if it won't eliminate all of the possible solutions
      - e.g. *n* queens – exploit symmetry: only consider rows 0..n/2-1 for the first queen placed
      - e.g. 0-1 knapsack – for process input, if the smallest remaining item doesn't fit in the pack, none of the others can be added either

pruning must be a local decision – cannot consider possible future scenarios of how specific choices would would play out in deciding whether or not to prune

Key Points – Making Backtracking Practical

- strategies

effective strategies tend to be highly problem-specific

  - *try locate good solutions quickly* – order the subproblems to explore more likely ones first
    - order the alternatives – consider the most promising alternative first
      - "most promising" must be a local decision
      - e.g. in maze solving via DFS, consider the adjacent rooms in order of straight-line distance to the goal (closest first)
      - e.g. in shortest path via DFS, consider the adjacent vertices in order of distance from the current vertex (closest first)
    - *best-first search*
      - put discovered subproblems in a priority queue ordered by cost of partial solution so far
    - *best-first search with A* heuristic*
      - put discovered subproblems in a priority queue ordered by estimated cost of the best complete solution derived from the partial solution

Branch and Bound

In optimization problems, pruning can also occur if no solutions in a subtree are good enough to be optimal.

Idea.

- have a function return the best possible cost of any solution derived from the current partial solution
  - if this cost is not better that the best known solution, the branch doesn't contain an optimal solution and can be pruned

Wrinkle.

- it is generally not possible to determine the best possible cost of a solution derived from a given partial solution without actually searching the subtree and finding all such solutions

Branch and Bound

Revised idea.

- have a function return *an estimate of* the best possible cost of any solution derived from the current partial solution
  - if the estimated cost is not better that the best known solution, the branch doesn't contain an optimal solution and it can be pruned

This *bound function* must be

- safe (optimistic about quality of solutions)
  - if the estimate is too optimistic (it claims better solutions than are possible), the only consequence is wasted time (unnecessary searching of a branch that doesn't actually have a better solution)
  - if the estimate is too pessimistic (it claims solutions are worse than they are), the consequence is possibly missing the optimal solution (thus getting the wrong answer)
- based only on the partial solution / current state
  - cannot consider how future choices play out – we're trying to *avoid* searching the subtree
- relatively efficient to compute
  - evaluated for every next choice

Branch and Bound

We also need to initialize the best known solution so there is something to compare the estimate to before the first solution is found.

- have a function return an estimate of the best possible cost of any solution

This function must be

- safe (pessimistic about quality of solutions)
  - if the estimate is too pessimistic (it claims the best solution is worse than it actually is), the only consequence is wasted time (unnecessary searching of branches with suboptimal solutions because they are thought to be better)
  - if the estimate is too optimistic (it claims the best solution is better than it actually is), the consequence is missing all solutions (thus getting the wrong answer) because no branch is thought to be good enough to be worth exploring
- more efficient than searching to compute

Branch and Bound

The practicality of a branch-and-bound algorithm depends on:

- the quality of the bound function
  - tighter bound means more and earlier pruning – but can't underestimate the true cost of the solutions in that subtree
- the initial solution value
  - closer to optimal means more and earlier pruning – but can't overestimate the cost of the actual best solution
- the time to compute the bound
  - better quality estimates are generally more expensive
  - expensive bound may not save enough work via pruning to be worthwhile – bound function must be computed for each alternative

(as well as the efficiency of the basic backtracking part – generating subproblems, updating partial solutions, pruning)

Branch and Bound

Design challenges:

- finding a good bound function
- finding a good initial solution value
- proving that clever bounds and initial solution values are safe

These tend to be highly problem-specific.

0-1 Knapsack

- frame as a series of choices – choose next item to take
- partial solution – set of items taken so far
- alternatives – items not yet chosen which fit in the pack
- subproblem – knapsack(S’,W')
  - input: set S’ of items left to consider, remaining unfilled capacity of the knapsack W'
  - output: items chosen, total value of those items
  - task: find the highest-value set of items whose total weight does not exceed the remaining capacity of the knapsack
- should we bother to solve knapsack(S’,W')?
  - pruning: no, if the smallest item in S’ exceeds W' nothing else will fit – the partial solution is actually a complete solution (treat as a base case)
  - branch and bound: no, if the best way to fill the pack from this point isn't better than the best solution already found
    - need an estimate of the best solution obtainable from this point
    - need an initial value for the best solution already found (until we find the first solution)

0-1 Knapsack

Bound function – upper bound or lower bound?

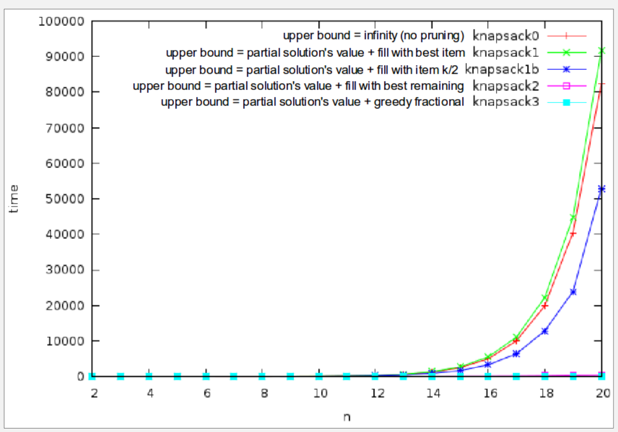
- maximization problem, so bigger is better
- prune if solutions aren't good enough → "safe" is an estimate which is too good → "too good" is bigger → looking for upper bound on the solution value

Which of the following would be safe choices for a bound function? Use the value of the items chosen so far plus ...

- [ ] filling the pack with the items will fit in their entirety (no fractional amounts), considered in order of decreasing value/weight ratio
- [ ] filling the pack with as much as possible of each of the remaining items (a fractional amount is allowed), in order of increasing value
- [ ] the lowest value/weight ratio of any item times the remaining capacity of the pack
- [ ] filling the pack with as much as possible of each of the remaining items (a fractional amount is allowed), in order of decreasing value
- [ ] the lowest value/weight ratio of any remaining (not yet considered) item times the remaining capacity of the pack
- [ ] filling the pack with as much as possible of each of the remaining items (a fractional amount is allowed), in order of increasing value/weight ratio
- ★ the highest value/weight ratio of any item times the remaining capacity of the pack
- ★ filling the pack with as much as possible of each of the remaining items (a fractional amount is allowed), in order of decreasing value/weight ratio
- [ ] filling the pack with the items will fit in their entirety (no fractional amounts), considered in order of increasing value/weight ratio
- ★ the highest value/weight ratio of any remaining (not yet considered) item times the remaining capacity of the pack

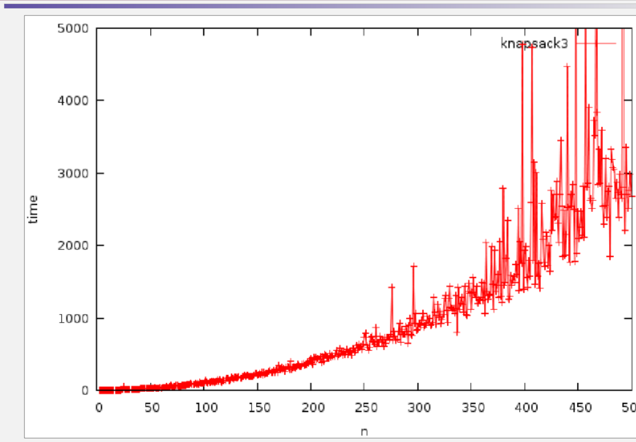
Bound Function Quality

range 1-100
capacity ~half of the total weight of all items



Bound Function Quality

(compare the range on the scales in this to the previous slide)



0-1 Knapsack

Initial solution value – upper bound or lower bound?

- maximization problem, so bigger is better
- update if solution is better → “safe” is an estimate which is not good enough → “not good enough” is smaller → looking for lower bound on the solution value

Which of the following would be safe choices for an initial solution estimate? Choose all that apply.

- ☆ consider the items in any order, taking each item if it fits in the pack
- ☆ 0
- ☐ the lowest value/weight ratio of any item times the capacity of the pack
- ☐ the highest value/weight ratio of any item times the capacity of the pack
- ☆ consider the items in increasing order of value/weight ratio, taking each item if it fits in the pack
- ☆ consider the items in decreasing order of value/weight ratio, taking each item if it fits in the pack

fill the pack with as much of each item as will fit (fractional amounts allowed), considering items in order of decreasing value/weight ratio

Additional Strategies

How much can be pruned depends on two things:

- the tightness of the bound function
- the value of the best solution so far, which depends on:
  - how well we can estimate its value at the beginning, and/or
  - how quickly a good solution is found

How to search the best branches first?

- modified depth-first search: at each step, order alternatives to explore most promising first
- best-first search: choose most promising subproblem first
  - priority queue stores discovered nodes of the search tree with priority corresponding to cost of partial solution
    - A* heuristic: use bound function (cost of any complete solution stemming from the partial solution)
  - downside of BFS is potentially exponential size of the queue

Bound Functions

Traveling Salesman Problem (TSP) – find a minimum cost cycle in a graph that visits every vertex exactly once.

- frame as a series of choices – which vertex to visit next
- partial solution
  - a path s → u
- subproblem – tsp(u,V')
  - find the min cost path from u using all vertices in V'
- alternatives
  - any vertex v' from V' such that (u,v') is an edge in the graph
- final answer
  - tsp(s,V-{s})
- complete solution
  - when V' is empty → solution is legal if there is an edge (u,s)
    - total cost of solution is partial solution cost + w(u,s)

Bound function – upper bound or lower bound? lower – minimization so low is good, thus optimistic is too low

Possible Bound Functions – TSP

- actual solution can't involve cheaper edges than the cheapest in the whole graph
  - bound: cost of path so far + cheapest edge in the whole graph × number of vertices remaining
  - safe but likely far from actual – cheapest edge can only be used once (and may have already been used)
- actual solution can't involve cheaper edges than the cheapest available
  - bound: cost of path so far + cheapest edge involving at least one vertex not yet in path × number of vertices remaining
  - safe but may be far from actual – cheapest edge can only be used once

Possible Bound Functions – TSP

- actual solution must contain an edge leaving u, and that can't be cheaper than the cheapest edge leaving u
  - bound: cost of path so far + cheapest next edge × number of vertices remaining
  - **not safe** – there may be cheaper edges available elsewhere for the rest of the path to use
- actual solution must contain an edge leaving u, and then the next vertex, and then the next vertex…
  - bound: cost of path so far + greedy solution
  - **not safe** – greedy solution may be worse than the best solution, but our estimate needs to be better (or at least equal)
- actual solution must contain at least an edge leaving u
  - bound: cost of path so far + cheapest next edge
  - safe (can't do better than the best edge leaving u) but very far from actual solution if partial solution is far from complete

Maximum Independent Set

Maximum independent set – find the largest subset of vertices in a graph such that no two in the set are connected by an edge. (sounds like the friend-of-my-friend, enemy-of-my-enemy problem…)

- frame as a series of choices – the next vertex to include
- partial solution
  - a set of vertices
- subproblem – maxindset(S)
  - S = S-v-{ u | (v,u) in G }, where v is the vertex chosen in this step
    - i.e. eliminate v and v's neighbors from S
- alternatives
  - each of the vertices v in S
    - note that it is not necessary to consider whether v is adjacent to an already-picked vertex
- final answer
  - maxindset(V)
- complete solution
  - when S is empty

Bound function – upper bound or lower bound?

upper – maximization so high is good, thus optimistic is too high

Possible Bound Functions – Maximum Ind Set

- can't do worse than an actual independent set of S
  - bound: number of vertices picked so far + a maximal independent set of S
    - "maximal" = can't be made larger by adding another vertex
      - can be found with a greedy algorithm: repeatedly take a legal vertex until there are no more
  - **not safe** – size of maximal independent set ≤ maximum
    - need upper bound (optimistic)

Possible Bound Functions – Maximum Ind Set

- can't do better than picking all remaining vertices
  - bound: number of vertices picked so far + |S|
  - safe, but not tight if there are very many edges
- at least some of those vertices are bound to be connected to each other…
  - picking one excludes its neighbors
    - bound: number of vertices picked so far + |S|-mindeg(S)
      - mindeg(S) = min degree of a vertex in S in the subgraph induced by S
      - **safe** – picking a vertex of S excludes its neighbors from consideration, and we can't exclude fewer than the min degree of one of those vertices
  - we can only choose one vertex per edge
    - bound: number of vertices picked so far + |S|/2
    - **not safe** – in a graph consisting of only isolated vertices, the maximum independent set has size |S|
  - ...refine the counting…
    - bound: number of vertices picked so far + $(1+\sqrt{(1-8m-4n+4n^2)})/2$
      - where the subgraph induced by the vertices in S has n vertices and m edges
    - **safe**

Strategies

A good bound function depends on the specific nature of the problem and what you can exploit about its structure.

But we've seen a few general tactics that might serve as starting points –

- value so far + best single choice ✕ number of choices left
- value so far + best single next choice ✕ number of choices left
  - only safe if all choices are available at each stage (e.g. knapsack but not TSP)
- value so far + greedy solution from that point
  - only safe if greedy can do better than the actual solution (true for knapsack, not for TSP and max independent set)
- consider trivial bound and what is over/undercounted
  - e.g. max independent set - |S| overcounts because a vertex and its neighbor can't both be in the set; |S|-mindeg(S) addresses that for one vertex picked

Initial Solution Estimate

Upper or lower bound?

- safe = conservative = worse than the optimal
  - if your estimate is better than the optimal, you'll prune away the branch containing the optimal as not good enough

Note: bound is on the value of the optimal solution, not the value of any legal solution

- e.g. “upper bound” does not mean that it needs to be worse than all possible legal solutions – and that wouldn't help you prune anything at all

Initial Solution Estimate

Any legal solution is a safe estimate – it will be no better than the optimal.

- greedy can be a good strategy
  - e.g. greedy TSP – take cheapest edge to not-yet-included vertex
  - e.g. maximal independent set – take any legal vertex until there are no more

But you may be able to get a tighter estimate without having an actual solution in mind.

(Then safety is important to establish.)

- e.g. 2*MST ≥ optimal TSP solution