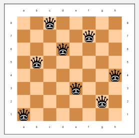
The Effectiveness of Reducing Search

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

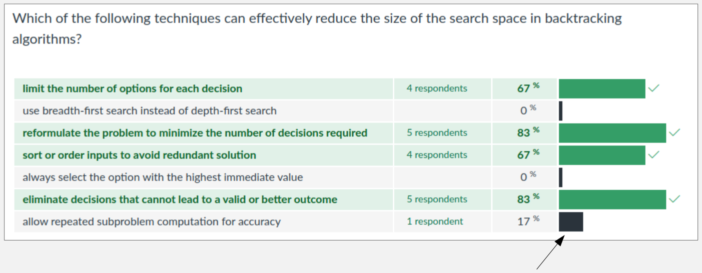
Key Points – Making Backtracking Practical

- 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



Which of the following techniques can effectively reduce the size of the search space in backtracking algorithms?

limit the number of options for each decision	4 respondents	**67 %**	✓
use breadth-first search instead of depth-first search		0 %	
reformulate the problem to minimize the number of decisions required	5 respondents	**83 %**	✓
sort or order inputs to avoid redundant solution	4 respondents	**67 %**	✓
always select the option with the highest immediate value		0 %	
eliminate decisions that cannot lead to a valid or better outcome	5 respondents	**83 %**	✓
allow repeated subproblem computation for accuracy	1 respondent	17 %	

caching solutions for repeated subproblems can very effectively reduce the size of the search space, but repeating computations for the same subproblems does the opposite

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

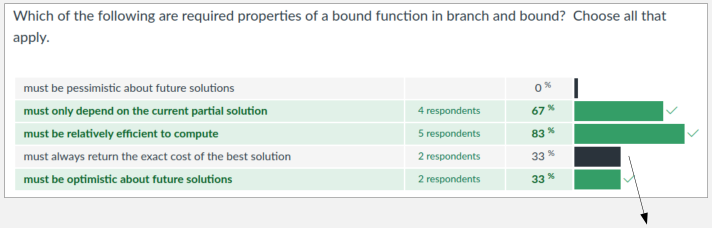
Branch and Bound

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

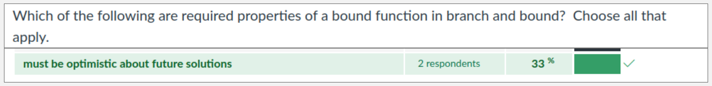
- safe (optimistic about quality of solutions)
  - if the estimate is too optimistic (it claims better solutions than are possible), the only consequence is wasted time (unnecessary searching of a branch that doesn't actually have a better solution)
  - if the estimate is too pessimistic (it claims solutions 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



Which of the following are required properties of a bound function in branch and bound? Choose all that apply.

must be pessimistic about future solutions		0 %	
must only depend on the current partial solution	4 respondents	**67 %**	✓
must be relatively efficient to compute	5 respondents	**83 %**	✓
must always return the exact cost of the best solution	2 respondents	33 %	
must be optimistic about future solutions	2 respondents	**33 %**	✓

would be ideal – but have to actually solve the rest of the problem for that

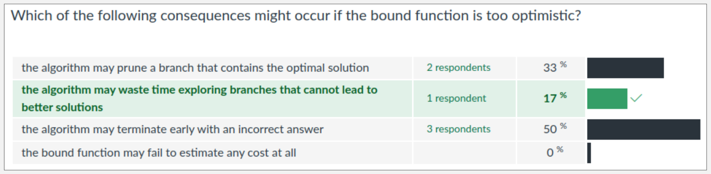


Which of the following are required properties of a bound function in branch and bound? Choose all that apply.

must be optimistic about future solutions	2 respondents	**33 %**	✓

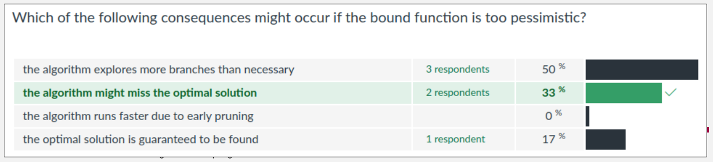
– prune if we can't improve on the best so far
– bound function returns an estimate of what can be done from here

– if that estimate is too good, the possible solutions look better than they are, so we keep pursuing them
– if that estimate is too poor, we'll abandon possible solutions that might actually turn out to be the ones we want



Which of the following consequences might occur if the bound function is too optimistic?

the algorithm may prune a branch that contains the optimal solution	2 respondents	33 %	
the algorithm may waste time exploring branches that cannot lead to better solutions	1 respondent	**17 %**	✓
the algorithm may terminate early with an incorrect answer	3 respondents	50 %	
the bound function may fail to estimate any cost at all		0 %	



Which of the following consequences might occur if the bound function is too pessimistic?

the algorithm explores more branches than necessary	3 respondents	50 %	
the algorithm might miss the optimal solution	2 respondents	**33 %**	✓
the algorithm runs faster due to early pruning		0 %	
the optimal solution is guaranteed to be found	1 respondent	17 %	

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.

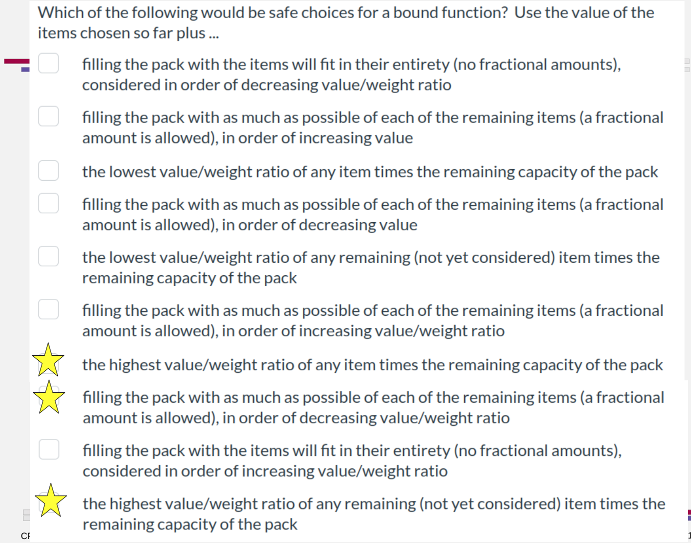
0-1 Knapsack

- 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

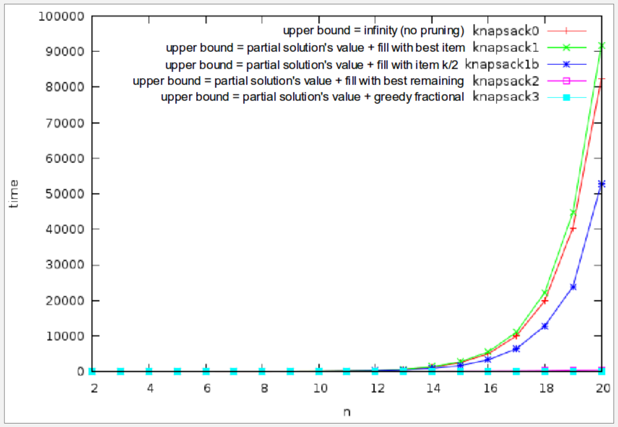
★ filling the pack with as much as possible of each of the remaining items (a fractional amount is allowed), in order of decreasing value/weight ratio

□ 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

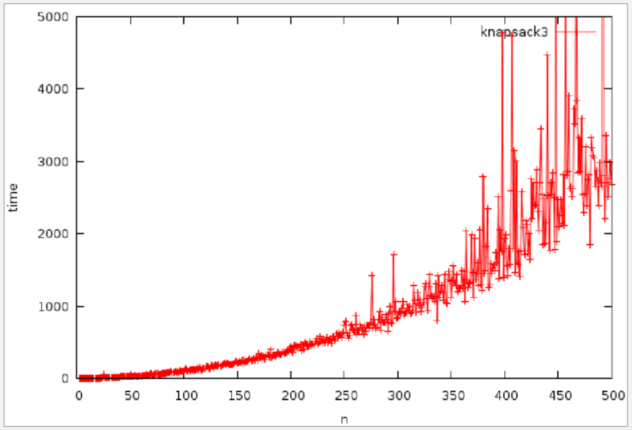
Bound Function Quality

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)



Strategies

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

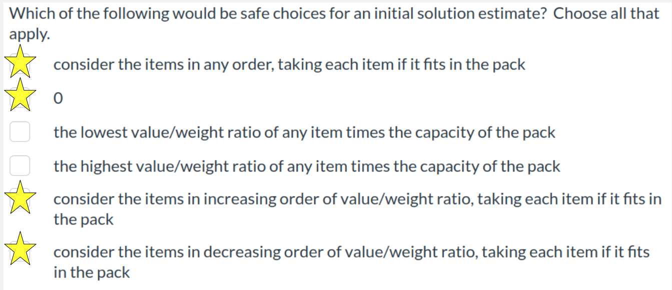
But we've seen a few general tactics that might serve as starting points –

- 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

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

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

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