• problems in the class NP are important to a number of applications
  • solution is of interest
  • NP problems do not have polynomial time algorithms that can produce an exact solution
    • must consider alternative algorithms that can produce reasonably good answers
    • some cases, algo will find optimal answer, but cannot be guaranteed
  • discussed depth-first and breadth-first traversal methods
    • DF uses recursion and stacks
      • Backtracking algorithms rely on recursion
      • make recursive call when there is a choice
        • saves state
        • if choice fails, can return
    • BF uses queue
      • Branch and Bound use a queue to hold problem states
        • have promise to solution
        • (removed from queue and examined)
        • promising choices added to queue

• probabilistic algorithms: sometimes better to guess than to figure out correct option
  • four classifications of probabilistic algos:
    • 1) numerical
    • 2) Monte Carlo
    • 3) Las Vegas
    • 4) Sherwood
  • common characteristic => produce better results the longer they run

• dynamic programming
  • keep track of previous information
  • e.g., fibonacci

• 11.1 Greedy Algorithms
  • Have seen greedy algorithms: e.g., minimum spanning tree algo, shortest path
    • look at number of greedy algos to approximate optimal solution for problems in NP
      • again, may not find exact, optimal solution
    • look at approximation algorithms for problems discussed:

• 11.1.1 Traveling salesman approximations
  • cannot just apply shortest path algorithm to solve this
    • can use Djikstra’s to find shortest path b/t two nodes,
      • but doesn’t visit every node in graph
    • we can use this general greedy technique to find an approximate algorithm
      • cost of traveling b/t cities represented by adj matrix
      • E.g.,
Algorithm:
- go through set of edges, pick them in order of increasing weight
- not concerned about forming a path
- make sure edges added to path meet 2 criteria:
  1) do not form a cycle with other edges (unless all nodes chosen)
  2) not the third edge connected to some node

  e.g.,
  - first choose edge (3,5) w/ weight of 1
  - then choose (5,6)
  - next edge would be (3,6) but reject b/c forms a cycle that is not complete
  - add edge (4,7)
  - and (2,7)
  - next edge would be (1,5) but rejected because tried edge to node 5
  - add edge (1,6)
  - then (1,4)
  - last edge (2,3)
  - gives us path 1, 4, 7, 2, 3, 5, 6, 1 w/ length 53..
  - optimal path:
    - path of 1, 4, 7, 2, 5, 3, 6, 1 w/ length 41

11.1.2 Bin Packing
- technique to approximate bin packing?
  - use first fit
    - for each object: look at the bins in order until one is found that has enough space to hold the object.
  - E.g.,
    - set of objects with sizes
      - \{ 0.5, 0.7, 0.3, 0.9, 0.6, 0.8, 0.1, 0.4, 0.2, 0.5 \}
    - how to pack bins?
      - bin 1: [ 0.5, 0.3, 0.1 ]
      - bin 2: [ 0.7, 0.1 ]
      - bin 3: [ 0.9 ]
      - bin 4: [ 0.6, 0.4 ]
      - bin 5: [ 0.8 ]
      - bin 6: [ 0.5 ]
    - optimal solution?
      - 5 bins
        - bin 1: [ 0.9, 0.1 ]
        - bin 2: [ 0.8, 0.2 ]
        - bin 3: [ 0.7, 0.3 ]
        - bin 4: [ 0.6, 0.4 ]
        - bin 5: [ 0.5, 0.5 ]
another approximation?

**decreasing first fit**
- objects first sorted in decreasing order
- then begin first fit
  - sometimes better than regular first fit
  - sometimes optimal
  - sometimes worse than regular first fit!
  - (can come up with counter example)

**11.1.3 Backpack approximation**
- create a sorted list of objects based on the ratio of worth to object size
- E.g.,
  - objects = { [25, 50], [20, 80], [20, 50], [15, 45], [30, 105], [35, 35], [20, 10], [10,45] }
  - worth ratios: (2, 4, 2.5, 3, 3.5, 1, 0.5, 4.5 )
  - sorting by worth ratio => [10,45], [20, 80], ...
- begin filling backpack using objects in the order of the sorted list
  - if the next object will not fit, skip and continue down the list.
- e.g., backpack of size 80,
  - able to fit first 4 objects for total size of 75 and worth of 275
  - not optimal.. 1st three and 5th object would yield 80 and 280.

**11.1.4 Subset Sum**
- similar to backpack..
- skipped...

**11.1.5 Graph coloring**
- unique problem
  - has been shown that getting an approximate coloring that is close to optimal is as complex as getting the optimal coloring.
  - best approx algorithm that runs in poly time will use more than twice as many colors as optimal!
  - has been shown that if there was a poly time algo that could color any graph with no more than twice as many as optimal,
    - then P = NP
- simple approximation:
  - use sequential coloring method
  - `sequentialColoring( G )`
    - // G is graph to be colored
    - for i = 1 to N
      - c = 1
      - while there is a node in G adjacent to node_i that is colored c, do
        - c = c + 1
      - color node_i with c
- approach will use C colors where C is 1 greater than the degree of the graph
  - (degree is largest number of edges leaving one node)
  - possible to do better, but pretty darn advanced.. (beyond scope of class)
11.2 Backtracking
- E.g., solving a maze
  - note whenever you make a choice between two directions
  - if you end up at a dead end
    - return to your choice and take a different path
    - return further back once all paths exhausted.
  - this technique to solve a maze is called **backtracking**
  - problem state is saved whenever a choice is made
  - if choice does not lead to solution,
    - restore problem state
    - try another choice
- Consider N-Queens
  - How to place N queens on an N×N board such that no queen will attack another.
  - 4 is smallest value of N for which there is a solution
  - can develop state-space tree that describes 4-Queen problem
    - can never been more than one queen per row or column
  - in state space tree,
    - each level represents possible places where each queen can be placed for one row on the board
    - root will have 4 children
      - => 1 node for each column
      - each child will have 4 nodes representing 4 columns where queen can be placed in row 2
        - not concerned with whether placement makes sense.
        - just describing entire problem space
  - path from root to leave gives us placement of four queens on the board
    - 256 leaves in tree
    - part of state space tree:
      - path with arrows shows route to solution
- we could use **depth first search** to solve the 4-queens problem
  - when a leaf is reached, the board will be checked for validity
  - a lot more work is done than necessary
    - we search part of tree that are totally invalid
    - E.g., going from 1,1 to 2,1, => need to go any further. No it’s invalid
      - saves us time.
    - **essence of how backtracking backtracking differs from DFS**
      - If a node in state-space tree could **never** lead to a solution,
then we consider it **nonpromising**
backtracking stops recursing when it reaches a nonpromising state

Algorithm:
- \texttt{Queens}(\textit{i})
  - if \textit{i} > \textit{N} then
    - output board
  - else
    - for \textit{j} = 1 to \textit{N}
      - \texttt{board}[\textit{i}] = \textit{j}
      - if no attacks
        - \texttt{Queens}(i+1)

Generic form of algorithm:
- \texttt{SearchSpace}(\textit{i}):
  - if there is a solution
    - output the solution
  - else
    - for every possible next step:
      - if the step is promising:
        - \texttt{SearchSpace}(i+1)

key is determining whether a state is promising.
- easy for N-Queens
- more complex for optimization problems

- How can **backtracking** be applied to Sudoku?

**11.3 Branch and Bound**
- similar to backtracking, but uses BFS instead of DFS
  - uses a queue instead of a stack
- B & B calculates a number called a bound
  - use it to determine if node is promising
    - I.e., if bound is not better than the best solution so far
      - no need to expand the node
      - cannot lead to improved solution
- used for optimization problems
  - “better” might mean larger solution to max problems
    - e.g., backpack
  - “better” => smaller solution for job scheduling
- can be improved by using priority queue..
- E.g., backpack problem
  - details omitted..
    - bounds => possible total size and worth
      - **non promising** if new size brings total over limit
      - also **non promising** if new worth is not greater than the best worth so far.
        - => those nodes will be omitted..