There are two principal approaches to tackling difficult
combinatorial problems (NP-hard problems):

- Use a strategy that guarantees solving the problem exactly
  but doesn’t guarantee to find a solution in polynomial time

- Use an approximation algorithm that can find an
  approximate (sub-optimal) solution in polynomial time
Exact Solution Strategies

- **exhaustive search** (brute force)
  - useful only for small instances

- **dynamic programming**
  - applicable to some problems (e.g., the knapsack problem)

- **backtracking**
  - eliminates some unnecessary cases from consideration
  - yields solutions in reasonable time for many instances but worst case is still exponential

- **branch-and-bound**
  - further refines the backtracking idea for optimization problems
Backtracking

- Construct the *state-space tree*
  - nodes: partial solutions
  - edges: choices in extending partial solutions

- Explore the state space tree using depth-first search

- “Prune” *nonpromising nodes*
  - stop exploring subtrees rooted at nodes that cannot lead to a solution and backtracks to such a node’s parent to continue the search
Example: $n$-Queens Problem

Place $n$ queens on an $n$-by-$n$ chess board so that no two of them are in the same row, column, or diagonal.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

queen 1
queen 2
queen 3
queen 4
State-Space Tree of the 4-Queens Problem
Example: Hamiltonian Circuit Problem

A Hamiltonian circuit is a cycle that visits every vertex exactly once.

Consider the graph with vertices a, b, c, d, e, f and edges:
- (a, b)
- (b, c)
- (c, d)
- (d, e)
- (e, f)
- (f, a)

The weights of the edges are:
- (a, b): 0
- (b, c): 0
- (c, d): 5
- (d, e): 3
- (e, f): 3
- (f, a): 6

We want to find a Hamiltonian circuit with minimum weight.

The solution to this problem is:
- (a, b, c, d, e, f, a)
- Minimum weight: 0 + 0 + 5 + 3 + 3 + 6 = 21

Note: The solution is found by considering all possible Hamiltonian circuits and selecting the one with the minimum weight.
Branch-and-Bound

- An enhancement of backtracking
- Applicable to optimization problems

For each node (partial solution) of a state-space tree, computes a bound on the value of the objective function for all descendants of the node (extensions of the partial solution)

Uses the bound for:
- ruling out certain nodes as “nonpromising” to prune the tree – if a node’s bound is not better than the best solution seen so far
- guiding the search through state-space
Example: Assignment Problem

Select one element in each row of the cost matrix $C$ so that:

- no two selected elements are in the same column
- the sum is minimized

Example

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person $a$</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Person $b$</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Person $c$</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Person $d$</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

Lower bound: Any solution to this problem will have total cost at least: $2 + 3 + 1 + 4$ (or $5 + 2 + 1 + 4$)
Example: First two levels of the state-space tree

Figure Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node's fields indicate the job number assigned to person a and the lower bound value, lb, for this node.
Example (cont.)

Figure  Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm
Example: Complete state-space tree

Figure Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm
Example: Traveling Salesman Problem
Approximation Approach

Apply a fast (i.e., a polynomial-time) approximation algorithm to get a solution that is not necessarily optimal but hopefully close to it.

Accuracy measures:

**Accuracy ratio** of an approximate solution $s_a$

$$r(s_a) = \frac{f(s_a)}{f(s^*)} \text{ for minimization problems}$$

$$r(s_a) = \frac{f(s^*)}{f(s_a)} \text{ for maximization problems}$$

where $f(s_a)$ and $f(s^*)$ are values of the objective function $f$ for the approximate solution $s_a$ and actual optimal solution $s^*$

**Performance ratio** of the algorithm $A$

the lowest upper bound of $r(s_a)$ on all instances
**Nearest-Neighbor Algorithm for TSP**

Starting at some city, always go to the nearest unvisited city, and, after visiting all the cities, return to the starting one

```
A -- 1 -- B
  
6   3   3   2
D  3   2   1   C
```

- $s_a : A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$ of length 10
- $s^* : A \rightarrow B \rightarrow D \rightarrow C \rightarrow A$ of length 8

**Note:** Nearest-neighbor tour may depend on the starting city

**Accuracy:** $R_A = \infty$ (unbounded above) – make the length of AD arbitrarily large in the above example
Multifragment-Heuristic Algorithm

Stage 1: Sort the edges in nondecreasing order of weights. Initialize the set of tour edges to be constructed to empty set.

Stage 2: Add next edge on the sorted list to the tour, skipping those whose addition would’ve created a vertex of degree 3 or a cycle of length less than $n$. Repeat this step until a tour of length $n$ is obtained.

Note: $R_A = \infty$, but this algorithm tends to produce better tours than the nearest-neighbor algorithm.
Twice-Around-the-Tree Algorithm

Stage 1: Construct a minimum spanning tree of the graph (e.g., by Prim’s or Kruskal’s algorithm)

Stage 2: Starting at an arbitrary vertex, create a path that goes twice around the tree and returns to the same vertex

Stage 3: Create a tour from the circuit constructed in Stage 2 by making shortcuts to avoid visiting intermediate vertices more than once

Note: $R_A = \infty$ for general instances, but this algorithm tends to produce better tours than the nearest-neighbor algorithm
Example

Walk: a → b → c → b → d → e → d → b → a

Tour: a → b → c → d → e → a
Christofides Algorithm

Stage 1: Construct a minimum spanning tree of the graph

Stage 2: Add edges of a minimum-weight matching of all the odd vertices in the minimum spanning tree

Stage 3: Find an Eulerian circuit of the multigraph obtained in Stage 2

Stage 3: Create a tour from the path constructed in Stage 2 by making shortcuts to avoid visiting intermediate vertices more than once

$R_A = \infty$ for general instances, but it tends to produce better tours than the twice-around-the-minimum-tree alg.
Example: Christofides Algorithm
Euclidean Instances

**Theorem** If $P \neq NP$, there exists no approximation algorithm for TSP with a finite performance ratio.

**Definition** An instance of TSP is called *Euclidean*, if its distances satisfy two conditions:

1. *symmetry* $d[i, j] = d[j, i]$ for any pair of cities $i$ and $j$
2. *triangle inequality* $d[i, j] \leq d[i, k] + d[k, j]$ for any cities $i, j, k$

For Euclidean instances:

- approx. tour length / optimal tour length $\leq 0.5(\lceil \log_2 n \rceil + 1)$ for nearest neighbor and multifragment heuristic;
- approx. tour length / optimal tour length $\leq 2$ for twice-around-the-tree;
- approx. tour length / optimal tour length $\leq 1.5$ for Christofides
Local Search Heuristics for TSP

Start with some initial tour (e.g., nearest neighbor). On each iteration, explore the current tour’s neighborhood by exchanging a few edges in it. If the new tour is shorter, make it the current tour; otherwise consider another edge change. If no change yields a shorter tour, the current tour is returned as the output.

Example of a 2-change
Example of a 3-change
### Empirical Data for Euclidean Instances

**TABLE 12.1** Average tour quality and running times for various heuristics on the 10,000-city random uniform Euclidean instances [Joh02]

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>% excess over the Held-Karp bound</th>
<th>Running time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>nearest neighbor</td>
<td>24.79</td>
<td>0.28</td>
</tr>
<tr>
<td>multifragment</td>
<td>16.42</td>
<td>0.20</td>
</tr>
<tr>
<td>Christofides</td>
<td>9.81</td>
<td>1.04</td>
</tr>
<tr>
<td>2-opt</td>
<td>4.70</td>
<td>1.41</td>
</tr>
<tr>
<td>3-opt</td>
<td>2.88</td>
<td>1.50</td>
</tr>
<tr>
<td>Lin-Kernighan</td>
<td>2.00</td>
<td>2.06</td>
</tr>
</tbody>
</table>
Greedy Algorithm for Knapsack Problem

Step 1: Order the items in decreasing order of relative values:
\[ \frac{v_1}{w_1} \geq \cdots \geq \frac{v_n}{w_n} \]

Step 2: Select the items in this order skipping those that don’t fit into the knapsack

Example: The knapsack’s capacity is 16

<table>
<thead>
<tr>
<th>item</th>
<th>weight</th>
<th>value</th>
<th>v/w</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>$40</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>$30</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>$50</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>$10</td>
<td>2</td>
</tr>
</tbody>
</table>

Accuracy

- \( R_A \) is unbounded (e.g., \( n = 2, \ C = m, \ w_1 = 1, \ v_1 = 2, \ w_2 = m, \ v_2 = m \))
- \( R_A \) yields exact solutions for the continuous version
Approximation Scheme for Knapsack Problem

Step 1: Order the items in decreasing order of relative values:
\[ \frac{v_1}{w_1} \geq \cdots \geq \frac{v_n}{w_n} \]

Step 2: For a given integer parameter \( k \), \( 0 \leq k \leq n \), generate all subsets of \( k \) items or less and for each of those that fit the knapsack, add the remaining items in decreasing order of their value to weight ratios.

Step 3: Find the most valuable subset among the subsets generated in Step 2 and return it as the algorithm’s output.

- **Accuracy**: \( \frac{f(s^*)}{f(s_a)} \leq 1 + \frac{1}{k} \) for any instance of size \( n \)
- **Time efficiency**: \( O(kn^{k+1}) \)
- **There are fully polynomial schemes**: algorithms with polynomial running time as functions of both \( n \) and \( k \)
**First-Fit (FF) Algorithm**: Consider the items in the order given and place each item in the first available bin with enough room for it; if there are no such bins, start a new one.

Example: \( n = 4, \ s_1 = 0.4, \ s_2 = 0.2, \ s_3 = 0.6, \ s_4 = 0.7 \)

**Accuracy**

- Number of extra bins never exceeds optimal by more than 70% (i.e., \( R_A \leq 1.7 \))
- Empirical average-case behavior is much better. (In one experiment with 128,000 bins, the relative error was found to be no more than 2%).
First-Fit Decreasing (FFD) Algorithm: Sort the items in decreasing order (i.e., from the largest to the smallest). Then proceed as above by placing an item in the first bin in which it fits and starting a new bin if there are no such bins.

Example: \( n = 4, \ s_1 = 0.4, \ s_2 = 0.2, \ s_3 = 0.6, \ s_4 = 0.7 \)

Accuracy

- Number of extra bins never exceeds optimal by more than 50% (i.e., \( R_A \leq 1.5 \))
- Empirical average-case behavior is much better, too
Numerical Algorithms

_Numerical algorithms_ concern with solving mathematical problems such as

- evaluating functions (e.g., $\sqrt{x}$, $e^x$, $\ln x$, $\sin x$)
- solving nonlinear equations
- finding extrema of functions
- computing definite integrals

Most such problems are of “continuous” nature and can be solved only approximately.
Principal Accuracy Metrics

- **Absolute error** of approximation (of $\alpha^*$ by $\alpha$)
  \[ |\alpha - \alpha^*| \]

- **Relative error** of approximation (of $\alpha^*$ by $\alpha$)
  \[ |\frac{\alpha - \alpha^*}{|\alpha^*|}| \]

  - undefined for $\alpha^* = 0$
  - often quoted in %
Two Types of Errors

- **truncation errors**
  - Taylor’s polynomial approximation
    \[
e^x \approx 1 + x + x^2/2! + \cdots + x^n/n!
    \]
    absolute error \(\leq M |x|^{n+1}/(n+1)!\) where \(M = \max e^t\) for \(0 \leq t \leq x\)
  - composite trapezoidal rule
    \[
    \int_a^b f(x)dx \approx (h/2) [f(a) + 2\sum_{1 \leq i \leq n-1} f(x_i) + f(b)], \quad h = (b - a)/n
    \]
    absolute error \(\leq (b-a)h^2 M_2 / 12\) where \(M_2 = \max |f''(x)|\) for \(a \leq x \leq b\)

- **round-off errors**
Solving Quadratic Equation

Quadratic equation $ax^2 + bx + c = 0$ ($a \neq 0$)

$$x_{1,2} = \frac{-b \pm \sqrt{D}}{2a} \text{ where } D = b^2 - 4ac$$

Problems:

- computing square root
  
  use Newton’s method: $x_{n+1} = 0.5(x_n + D/x_n)$

- subtractive cancellation
  
  use alternative formulas (see p. 411)
  
  use double precision for $D = b^2 - 4ac$

- other problems (overflow, etc.)
Notes on Solving Nonlinear Equations

There exist no formulas with arithmetic ops. and root extractions for roots of polynomials

\[ a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0 = 0 \quad \text{of degree } n \geq 5 \]

Although there exist special methods for approximating roots of polynomials, one can also use general methods for

\[ f(x) = 0 \]

Nonlinear equation \( f(x) = 0 \) can have one, many, infinitely many, and no roots at all

Useful:

- sketch graph of \( f(x) \)
- separate roots
Three Classic Methods

Three classic methods for solving nonlinear equation

\[ f(x) = 0 \]

in one unknown:

- bisection method
- method of false position (regula falsi)
- Newton’s method
Bisection Method

Based on

- Theorem: If $f(x)$ is continuous on $a \leq x \leq b$ and $f(a)$ and $f(b)$ have opposite signs, then $f(x) = 0$ has a root on $a < x < b$

- Binary search idea

Approximations $x_n$ are middle points of shrinking segments

- $|x_n - x^*| \leq (b - a)/2^n$

- $x_n$ always converges to root $x^*$ but slower compared to others
Example of Bisection Method Application

Find the root of

\[ x^3 - x - 1 = 0 \]

with the absolute error not larger than 0.01

<table>
<thead>
<tr>
<th>( n )</th>
<th>( a_n )</th>
<th>( b_n )</th>
<th>( x_n )</th>
<th>( f(x_n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0-</td>
<td>2.0+</td>
<td>1.0</td>
<td>-1.0</td>
</tr>
<tr>
<td>2</td>
<td>1.0-</td>
<td>2.0+</td>
<td>1.5</td>
<td>0.875</td>
</tr>
<tr>
<td>3</td>
<td>1.0-</td>
<td>1.5+</td>
<td>1.25</td>
<td>-0.296875</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1.3125-</td>
<td>1.328125+</td>
<td>1.3203125</td>
<td>-0.018711</td>
</tr>
</tbody>
</table>

\[ x \approx 1.3203125 \]
Method of False Position

Similar to bisection method but uses $x$-intercept of line through $(a, f(a))$ and $(b, f(b))$ instead of middle point of $[a,b]$

Approximations $x_n$ are computed by the formula

$$x_n = \frac{a_n f(b_n) - b_n f(a_n)}{f(b_n) - f(a_n)}$$

Normally $x_n$ converges faster than bisection method sequence but slower than Newton’s method sequence
Newton’s Method

Very fast method in which $x_n$’s are $x$-intercepts of tangent lines to the graph of $f(x)$

Approximations $x_n$ are computed by the formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$
Notes on Newton’s Method

- Normally, approximations $x_n$ converge to root very fast but can diverge with a bad choice of initial approximation $x_0$

- Yields a very fast method for computing square roots
  $$x_{n+1} = 0.5(x_n + D/x_n)$$

- Can be generalized to much more general equations