Combinatorial Search

- permutations
- backtracking
- counting
- subsets
- paths in a graph

Overview

Exhaustive search. Iterate through all elements of a search space.

Applicability. Huge range of problems (include intractable ones).

Caveat. Search space is typically exponential in size ⇒ effectiveness may be limited to relatively small instances.


Warmup: enumerate N-bit strings

Goal. Process all $2^N$ bit strings of length N.
- Maintain $a[i]$ where $a[i]$ represents bit $i$.
- Simple recursive method does the job.

```java
// enumerate bits in a[k] to a[N-1]
private void enumerate(int k)
{
    if (k == N)
    {  process(); return;  }
    enumerate(k+1);
    a[k] = 1;
    enumerate(k+1);
    a[k] = 0;
}
```

Remark. Equivalent to counting in binary from 0 to $2^N - 1$.

N = 4

Warmup: enumerate N-bit strings

```java
public class BinaryCounter
{
    private int N;   // number of bits
    private int[] a; // a[i] = ith bit
    public BinaryCounter(int N)
    {
        this.N = N;
        this.a = new int[N];
        enumerate(0);
    }

    private void process()
    {
        for (int i = 0; i < N; i++)
            StdOut.print(a[i] + " ");
        StdOut.println();
    }

    private void enumerate(int k)
    {
        if (k == N)
        {  process(); return;  }
        enumerate(k+1);
        a[k] = 1;
        enumerate(k+1);
        a[k] = 0;
    }
}
```

public static void main(String[] args)
{
    int N = Integer.parseInt(args[0]);
    new BinaryCounter(N);
}

% java BinaryCounter 4
0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1

all programs in this lecture are variations on this theme
N-rooks problem

Q. How many ways are there to place N rooks on an N-by-N board so that no rook can attack any other?

Representation. No two rooks in the same row or column ⇒ permutation.

Challenge. Enumerate all N! permutations of 0 to N-1.

Enumerating permutations

Recursive algorithm to enumerate all N! permutations of size N.
• Start with permutation a[0] to a[N-1].
• For each value of i:
  - swap a[i] into position 0
  - enumerate all (N-1)! permutations of a[1] to a[N-1]
  - clean up (swap a[i] back to original position)

```java
int[] a = { 2, 0, 1, 3, 6, 7, 4, 5 };
```

// place N-k rooks in a[k] to a[N-1]
private void enumerate(int k)
{
    if (k == N)
    {  process(); return;  }
    for (int i = k; i < N; i++)
    {
        exch(k, i);
        enumerate(k+1);
        exch(i, k);
    }
}

```
```
public class Rooks
{
    private int N;
    private int[] a; // bits (0 or 1)

    public Rooks(int N)
    {
        this.N = N;
        a = new int[N];
        for (int i = 0; i < N; i++)
            a[i] = i;
        enumerate(0);
    }

    private void enumerate(int k)
    { /* see previous slide */ }

    private void exch(int i, int j)
    { int t = a[i]; a[i] = a[j]; a[j] = t; }

    public static void main(String[] args)
    { int N = Integer.parseInt(args[0]);
        new Rooks(N);
    }
}

N-rooks problem: back-of-envelope running time estimate

Slow way to compute N!.

Hypothesis. Running time is about 2(N! / 8!) seconds.
Q. How many ways are there to place \( N \) queens on an \( N \times N \) board so that no queen can attack any other?

Representation. No two queens in the same row or column ⇒ permutation. Additional constraint. No diagonal attack is possible.

Challenge. Enumerate (or even count) the solutions. Unlike \( N \)-rooks problem, nobody knows answer for \( N > 30 \).

N-queens problem

int[] a = { 2, 7, 3, 6, 0, 5, 1, 4 };  

4-queens search tree

Backtracking paradigm. Iterate through elements of search space.
• When there are several possible choices, make one choice and recur.
• If the choice is a dead end, backtrack to previous choice, and make next available choice.

Benefit. Identifying dead ends allows us to prune the search tree.

Ex. [backtracking for \( N \)-queens problem]
• Dead end: a diagonal conflict.
• Pruning: backtrack and try next column when diagonal conflict found.
private boolean backtrack(int k)
{
    for (int i = 0; i < k; i++)
    {
        if ((a[i] - a[k]) == (k - i)) return true;
        if ((a[k] - a[i]) == (k - i)) return true;
    }
    return false;
}

// place N-k queens in a[k] to a[N-1]
private void enumerate(int k)
{
    if (k == N) { process(); return; }
    for (int i = k; i < N; i++)
    {
        exch(k, i);
        if (!backtrack(k)) enumerate(k+1);
        exch(i, k);
    }
}

N-queens problem: backtracking solution

Pruning the search tree leads to enormous time savings.

N-queens problem: effectiveness of backtracking

Hypothesis. Running time is about \( (N! / 2.5^N) / 43,000 \) seconds.

Conjecture. \( Q(N) \approx N! / c^N \), where \( c \) is about 2.54.
Counting: Java implementation

**Goal.** Enumerate all N-digit base-R numbers.

**Solution.** Generalize binary counter in lecture warmup.

```java
define base-R numbers in a[k] to a[N-1]
private static void enumerate(int k) {
    if (k == N) {
        process(); return;
    }
    for (int r = 0; r < R; r++) {
        a[k] = r;
        enumerate(k+1);
    }
    a[k] = 0;
}
```

Counting application: Sudoku

**Goal.** Fill 9-by-9 grid so that every row, column, and box contains each of the digits 1 through 9.

**Solution.** Enumerate all 81-digit base-9 numbers (with backtracking).

```
<table>
<thead>
<tr>
<th>7</th>
<th>2</th>
<th>8</th>
<th>9</th>
<th>4</th>
<th>6</th>
<th>3</th>
<th>1</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>3</td>
<td>8</td>
<td>2</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>7</td>
<td>5</td>
<td>9</td>
<td>3</td>
<td>8</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>9</td>
<td>4</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>4</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>3</td>
<td>6</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>9</td>
<td>5</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>5</td>
<td>8</td>
<td>2</td>
<td>4</td>
<td>9</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Remark. Natural generalization is NP-complete.

Sudoku: backtracking solution

Iterate through elements of search space.
- For each empty cell, there are 9 possible choices.
- Make one choice and recur.
- If you find a conflict in row, column, or box, then backtrack.
private void enumerate(int k) {
    if (k == 81) {
        process(); return;
    }
    if (a[k] != 0) {
        enumerate(k+1); return;
    }
    for (int r = 1; r <= 9; r++) {
        a[k] = r;
        if (!backtrack(k)) enumerate(k+1);
    }
    a[k] = 0;
}

private void enumerate(int k) {
    if (k == N) {
        process(); return;
    }
    enumerate(k+1);
    a[k] = 1;
    enumerate(k+1);
    a[n] = 0;
}

Enumerating subsets: natural binary encoding

Given N items, enumerate all $2^N$ subsets.
- Count in binary from 0 to $2^N - 1$.
- Bit $i$ represents item $i$.
- If 0, in subset; if 1, not in subset.

<table>
<thead>
<tr>
<th>$i$</th>
<th>binary</th>
<th>subset</th>
<th>complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 0</td>
<td>empty</td>
<td>4 3 2 1</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 1</td>
<td>1</td>
<td>4 3 2</td>
</tr>
<tr>
<td>2</td>
<td>0 0 1 0</td>
<td>2</td>
<td>4 3 1</td>
</tr>
<tr>
<td>3</td>
<td>0 0 1 1</td>
<td>2 1</td>
<td>4 3</td>
</tr>
<tr>
<td>4</td>
<td>0 1 0 0</td>
<td>3</td>
<td>4 2 1</td>
</tr>
<tr>
<td>5</td>
<td>0 1 0 1</td>
<td>3 1</td>
<td>4 2</td>
</tr>
<tr>
<td>6</td>
<td>0 1 1 0</td>
<td>3 2</td>
<td>4 1</td>
</tr>
<tr>
<td>7</td>
<td>0 1 1 1</td>
<td>3 2 1</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>1 0 0 0</td>
<td>4</td>
<td>3 2 1</td>
</tr>
<tr>
<td>9</td>
<td>1 0 0 1</td>
<td>4 1</td>
<td>3 2</td>
</tr>
<tr>
<td>10</td>
<td>1 0 1 0</td>
<td>4 2</td>
<td>3 1</td>
</tr>
<tr>
<td>11</td>
<td>1 0 1 1</td>
<td>4 2 1</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>1 1 0 0</td>
<td>4 3</td>
<td>2 1</td>
</tr>
<tr>
<td>13</td>
<td>1 1 0 1</td>
<td>4 3 1</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>1 1 1 0</td>
<td>4 3 2</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>1 1 1 1</td>
<td>4 3 2 1</td>
<td>empty</td>
</tr>
</tbody>
</table>
Digression: Samuel Beckett play

Quad. Starting with empty stage, 4 characters enter and exit one at a time, such that each subset of actors appears exactly once.

<table>
<thead>
<tr>
<th>code</th>
<th>subset</th>
<th>move</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>empty</td>
<td></td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>enter 1</td>
<td>1</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>enter 2</td>
<td>2</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>exit 1</td>
<td>3</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>enter 3</td>
<td>4</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>enter 4</td>
<td>5</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>exit 2</td>
<td>6</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>exit 3</td>
<td>7</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>enter 5</td>
<td>8</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>enter 6</td>
<td>9</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>enter 7</td>
<td>10</td>
</tr>
</tbody>
</table>

"faces, emotionless one of the far future, a world where people are born, go through prescribed movements, fear non-being even though their lives are meaningless, and then they disappear or die." — Sidney Homan

Binary reflected gray code

Def. The k-bit binary reflected Gray code is:
- the (k-1) bit code with a 0 prepended to each word, followed by
- the (k-1) bit code in reverse order, with a 1 prepended to each word.

Enumerating subsets using Gray code

Two simple changes to binary counter from warmup:
- Flip a[k] instead of setting it to 1.
- Eliminate cleanup.

```java
// all bit strings in a[k] to a[N-1]
private void enumerate(int k) {
    if (k == N) {
        process(); return;
    }
    enumerate(k+1);
    a[k] = 1;
    enumerate(k+1);
    a[k] = 0;
}
```

Gray code binary counter vs. standard binary counter (from warmup)

Advantage. Only one item in subset changes at a time.
More applications of Gray codes

- 3-bit rotary encoder
- Chinese ring puzzle
- 8-bit rotary encoder
- Towers of Hanoi

33

Scheduling

Scheduling (set partitioning). Given \( n \) jobs of varying length, divide among
two machines to minimize the makespan (time the last job finishes).

Remark. This scheduling problem is NP-complete.

34

Scheduling (full implementation)

```java
public class Scheduler {
    private int N; // Number of jobs.
    private int[] a; // Subset assignments.
    private int[] b; // Best assignment.
    private double[] jobs; // Job lengths.

    public Scheduler(double[] jobs) {
        this.N = jobs.length;
        this.jobs = jobs;
        a = new int[N];
        b = new int[N];
        enumerate(N);
    }

    public int[] best() {
        return b;
    }

    private void enumerate(int k) {
        /* Gray code enumeration. */
    }

    private void process() {
        if (cost(a) < cost(b)) {
            for (int i = 0; i < N; i++)
                b[i] = a[i];
        }
    }

    public static void main(String[] args) {
        /* create Scheduler, print results */
    }
}
```

Scheduling (larger example)

Observation. Large number of subsets
leads to remarkably low cost.
### Scheduling: improvements

Many opportunities (details omitted).

- Fix last job to be on machine 0 (quick factor-of-two improvement).
- Maintain difference in finish times (instead of recomputing from scratch).
- Backtrack when partial schedule cannot beat best known.
  (check total against goal: half of total job times)

```java
private void enumerate(int k)
{
    if (k == N-1)
    {
        process();
        return;
    }
    if (backtrack(k))
    {
        return;
    }
    enumerate(k+1);
    a[k] = 1 - a[k];
    enumerate(k+1);
}
```

- Process all $2^k$ subsets of last $k$ jobs, keep results in memory,
  (reduces time to $2^{N-k}$ when $2^k$ memory available).

---

### Enumerating all paths on a grid

**Goal.** Enumerate all simple paths on a grid of adjacent sites.

- **Application.** Self-avoiding lattice walk to model polymer chains.

**Boggle.** Find all words that can be formed by tracing a simple path of adjacent cubes (left, right, up, down, diagonal).

**Pruning.** Stop as soon as no word in dictionary contains string of letters on current path as a prefix ⇒ use a trie.
Boggle: Java implementation

private void dfs(String prefix, int i, int j) {
    if ((i < 0 || i >= N) ||
        (j < 0 || j >= N) ||
        visited[i][j])  ||
        !dictionary.containsAsPrefix(prefix))
        return;
    visited[i][j] = true;
    prefix = prefix + board[i][j];
    if (dictionary.contains(prefix))
        found.add(prefix);
    for (int ii = -1; ii <= 1; ii++)
        for (int jj = -1; jj <= 1; jj++)
            dfs(prefix, i + ii, j + jj);
    visited[i][j] = false;
}

Hamilton path

Goal. Find a simple path that visits every vertex exactly once.

Remark. Euler path easy, but Hamilton path is NP-complete.

Knight’s tour

Goal. Find a sequence of moves for a knight so that (starting from any desired square) it visits every square on a chessboard exactly once.

Solution. Find a Hamilton path in knight’s graph.

Hamilton path: backtracking solution

Backtracking solution. To find Hamilton path starting at \( v \):
- Add \( v \) to current path.
- For each vertex \( w \) adjacent to \( v \)
  - find a simple path starting at \( w \) using all remaining vertices
- Clean up: remove \( v \) from current path.

Q. How to implement?
A. Add cleanup to DFS (!)
Hamilton path: Java implementation

```java
public class HamiltonPath {
    private boolean[] marked; // vertices on current path
    private int count = 0;    // number of Hamiltonian paths

    public HamiltonPath(Graph G) {
        marked = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)
            dfs(G, v, 1);
    }

    private void dfs(Graph G, int v, int depth) {
        marked[v] = true;
        if (depth == G.V()) count++;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w, depth+1);
        marked[v] = false; // clean up
    }
}
```

Exhaustive search: summary

<table>
<thead>
<tr>
<th>problem</th>
<th>enumeration</th>
<th>backtracking</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-rooks</td>
<td>permutations</td>
<td>no</td>
</tr>
<tr>
<td>N-queens</td>
<td>permutations</td>
<td>yes</td>
</tr>
<tr>
<td>Sudoku</td>
<td>base-9 numbers</td>
<td>yes</td>
</tr>
<tr>
<td>scheduling</td>
<td>subsets</td>
<td>yes</td>
</tr>
<tr>
<td>Boggle</td>
<td>paths in a grid</td>
<td>yes</td>
</tr>
<tr>
<td>Hamilton path</td>
<td>paths in a graph</td>
<td>yes</td>
</tr>
</tbody>
</table>

That’s all, folks: Keep searching!