Agenda

1. Introduction
2. Uninformed Search versus Informed Search (Best First Search, A* Search, Heuristics)
3. Constraint Satisfaction
4. Structural Decomposition Techniques (Tree/Hypertree Decompositions)
5. Local Search, Stochastic Hill Climbing, Simulated Annealing
6. Tabu Search
7. Evolutionary Algorithms/ Genetic Algorithms
8. Adversial Search and Game Playing
Traditional Methods

- There are many classic algorithms to search spaces for an optimal solution.
- Broadly, they fall into two disjoint classes:
  - Algorithms that only evaluate complete solutions (exhaustive search, local search, ...).
  - Algorithms that require the evaluation of partially constructed or approximate solutions.
- Algorithms that treat complete solutions can be stopped any time, and give at least one potential answer.
- If you interrupt an algorithm that works on partial solutions, the results might be useless.
Complete Solutions

- All decision variables are specified.
- For example, binary strings of length $n$ constitute complete solutions for any $n$-variable SAT.
- Permutations of $n$ cities constitute complete solutions for a TSP.
- We can compare two complete solutions using an evaluation function.
- Many algorithms rely on such comparisons, manipulating one single complete solution at a time.
- When a new solution has a better evaluation than the previous best solution, it replaces that prior solution.
- Exhaustive search, local search, hill climbing as well as modern heuristic methods such as simulated annealing, tabu search and evolutionary algorithms fall into this category.
Partial Solutions

There are two forms:

1. incomplete solution to the problem originally posed, and
2. complete solution to a reduced (i.e. simpler) problem.

- Incomplete solutions reside in a subset of the original problem’s search space.
  - In an SAT, consider all of the binary strings where the first two variables were assigned the value 1 (i.e. TRUE).
  - In a TSP, consider every permutation of cities that contains the sequence 7 — 11 — 2 — 16.
  - We fix the attention on a subset of the search space that has a partial property.
  - Hopefully, that property is also shared by the real solution!
Partial Solutions ctd.

- Decompose original problem into a set of smaller and simpler problems.
  - Hope: solving each of the easier problems and combine the partial solutions, results in an answer for the original problem.
  - In a TSP, consider only $k$ out of $n$ cities and try to establish the shortest path from city $i$ to $j$ that passes through all $k$ of these cities.
  - Reduce the size of the search space significantly and search for a complete solution within the restricted domain.
  - Such partial solutions can serve as building blocks for the solution to the original problem.
Partial Solutions ctd.

- **Decompose** original problem into a set of **smaller** and **simpler** problems.
  - Hope: solving each of the easier problems and **combine the partial solutions**, results in an answer for the original problem.
  - In a TSP, consider only $k$ out of $n$ cities and try to establish the shortest path from city $i$ to $j$ that passes through all $k$ of these cities.
  - **Reduce the size of the search space** significantly and search for a complete solution within the restricted domain.
  - Such partial solutions can serve as **building blocks** for the solution to the original problem.

- But, algorithms that work on partial solutions pose **additional difficulties**. One needs to
  - devise a way to **organize the subspaces** so that they can be searched efficiently, and
  - create a **new evaluation function** that can assess the quality of partial solutions.
Exhaustive Search

- Checks every solution in the search space until the best global solution has been found.
- Can be used only for small instances of problems.
- Exhaustive (enumerative) algorithms are simple.
- Search space can be reduced by backtracking.
- Some optimization methods, e.g., branch and bound and A* are based on an exhaustive search.
Exhaustive Search

- Checks every solution in the search space until the best global solution has been found.
- Can be used only for small instances of problems.
- Exhaustive (enumerative) algorithms are simple.
- Search space can be reduced by backtracking.
- Some optimization methods, e.g., branch and bound and A* are based on an exhaustive search.
- How can we generate a sequence of every possible solution to the problem?
  - The order in which the solutions are generated and evaluated is irrelevant (because we evaluate all of them).
  - The answer for the question depends on the selected representation.
Enumerating the SAT

- We have to generate every possible binary string of length $n$.
- All solutions correspond to whole numbers in a one-to-one mapping.
- Generate all non-negative integers from 0 to $2^n - 1$ and convert each of these integers into the matching binary string of length $n$.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>1011</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>1101</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>1111</td>
<td>15</td>
</tr>
</tbody>
</table>

- Bits of the string are the truth assignments of the decision variables.
- Organize the search space, for example partition into two disjoint subspaces. First contains all the vectors where $x_1 = \text{f}$ (FALSE), and the second contains all vectors where $x_1 = \text{t}$ (TRUE).
Enumerating the SAT ctd.

Binary search tree for SAT
Enumerating the TSP

- How to generate all possible permutations?
- If some cities are not connected, some permutation might not be feasible.

**Algorithm 1 gen1_permutation(i)**

```
k ← k + 1
P[i] ← k
if k = n then
    for q = 1 to n do
        print P[q]
    end for
end if
for q = 1 to n do
    if P[q] = 0 then
        gen1_permutation(q)
    end if
end for
k ← k - 1
P[i] ← 0
```
Algorithm 2 gen1_permutation(i)

\[ k \leftarrow k + 1 \]
\[ P[i] \leftarrow k \]
if \( k = n \) then
  for \( q = 1 \) to \( n \) do
    print \( P[q] \)
  end for
end if
for \( q = 1 \) to \( n \) do
  if \( P[q] = 0 \) then
    gen1_permutation(q)
  end if
end for
\[ k \leftarrow k - 1 \]
\[ P[i] \leftarrow 0 \]

- Called with \( k \) initialized to \(-1\), parameter \( i \) set to \( 0 \), and all entries of the array \( P \) initialized to \( 0 \);
- Prints every permutation of \((1, \ldots, n)\).
- Fixes 1 in the first position and generates the remaining \((n - 1)!\) permutations of numbers 2 to \( n \).
- For \( n = 3 \): \((1 \ 2 \ 3), (1 \ 3 \ 2), (2 \ 1 \ 3), (3 \ 1 \ 2), (2 \ 3 \ 1), (3 \ 2 \ 1)\).
Search Strategies

A strategy is defined by picking the order of node expansion. Strategies are evaluated along the following dimensions:

- **Completeness** - does it always find a solution if one exists?
- **Time complexity** - number of nodes generated/expanded.
- **Space complexity** - maximum number of nodes in memory.
- **Optimality** - does it always find a least-cost solution?

Time and space complexity are measured in terms of

- \( b \) - maximum branching factor of the search tree;
- \( d \) - depth of the least-cost solution;
- \( m \) - maximum depth of the state space (may be \( \infty \)).
Uninformed Search Strategies

Uninformed strategies use only the information available in the problem definition.

- Breadth-first search
- Depth-first search
- Depth-limited search
- Iterative deepening search
Breadth-First Search

- Expand shallowest unexpanded node.
- FIFO queue, i.e. new nodes go to the back of the queue, and old nodes get expanded first.
Breadth-First Search

- Expand shallowest unexpanded node.
- FIFO queue, i.e. new nodes go to the back of the queue, and old nodes get expanded first.
Breadth-First Search

- Expand shallowest unexpanded node.
- FIFO queue, i.e. new nodes go to the back of the queue, and old nodes get expanded first.
Breadth-First Search

- Expand shallowest unexpanded node.
- FIFO queue, i.e. new nodes go to the back of the queue, and old nodes get expanded first.
Properties of breadth-first search

- **Complete**: Yes (if $b$ is finite)
- **Time**: $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in $d$
- **Space**: $O(b^{d+1})$ (keeps every node in memory)
- **Optimal**: Yes (if cost = 1 per step); not optimal in general

Space is the big problem; can easily generate nodes at 100MB/sec so 24hrs = 8640GB.
Depth-First Search

- Expand deepest unexpanded node.
- LIFO queue, i.e. most recently generated node is chosen for expansion.
Depth-First Search

- Expand deepest unexpanded node.
- LIFO queue, i.e. most recently generated node is chosen for expansion.
Depth-First Search

- Expand deepest unexpanded node.
- LIFO queue, i.e. most recently generated node is chosen for expansion.
Depth-First Search

- Expand deepest unexpanded node.
- LIFO queue, i.e. most recently generated node is chosen for expansion.
Depth-First Search

- Expand deepest unexpanded node.
- LIFO queue, i.e. most recently generated node is chosen for expansion.
Depth-First Search

- Expand deepest unexpanded node.
- LIFO queue, i.e. most recently generated node is chosen for expansion.
Depth-First Search

- Expand deepest unexpanded node.
- LIFO queue, i.e. most recently generated node is chosen for expansion.
Depth-First Search

- Expand deepest unexpanded node.
- LIFO queue, i.e. most recently generated node is chosen for expansion.

Diagram:

- Nodes A, B, C, D, E, F, G, H, I, J, K, L, M, N, O
- A expands E, which expands K
- B and C are not expanded
- Green nodes F, G, L, M, N, O are not part of the current search
Depth-First Search

- Expand deepest unexpanded node.
- LIFO queue, i.e. most recently generated node is chosen for expansion.
Depth-First Search

- Expand deepest unexpanded node.
- LIFO queue, i.e. most recently generated node is chosen for expansion.
Depth-First Search

- Expand deepest unexpanded node.
- LIFO queue, i.e. most recently generated node is chosen for expansion.
Depth-First Search

- Expand deepest unexpanded node.
- LIFO queue, i.e. most recently generated node is chosen for expansion.
Properties of Depth-First Search

**Complete**  No: fails in infinite-depth spaces, spaces with loops

**Time**  \(O(b^m)\): terrible if \(m\) is much larger than \(d\);
but if solutions are dense, may be much faster than breadth-first

**Space**  \(O(bm)\), i.e., linear space!

**Optimal**  No
Backtracking

Suppose the SAT formula $\varphi$ contains a clause $(x_1 \lor x_2)$.

\[ \begin{align*}
&\varphi \\
&x_1 = f \\
&x_2 = f \\
&x_3 = f \\
&x_1 = t \\
&x_2 = t \\
&x_3 = t \\
&x_2 = f \\
&x_1 = t \\
&x_2 = t \\
&x_3 = t \\
&\ldots \\
&\ldots
\end{align*} \]
Backtracking

Suppose the SAT formula $\varphi$ contains a clause $(x_1 \lor x_2)$.

$\varphi$

- $x_1 = f$
- $x_1 = t$
- $x_2 = f$
- $x_2 = t$
- $x_2 = f$
- $x_2 = t$
- $x_3 = f$
- $x_3 = t$

Remaining branches below this node can lead to nothing but a dead end.
Backtracking

Suppose the SAT formula $\varphi$ contains a clause $(x_1 \lor x_2)$.

Remaining branches below this node can lead to nothing but a dead end.
Depth-Limited Search

• Depth first search with depth limit L
  – Nodes at depth L are not expanded.
• Eliminates problem with infinite path.
• How to select L?
• Possible failures:
  – No solution;
  – Cutoff - no solution within the depth limit.
Iterative Deepening Search

Repeat Depth-limited search with $L=1,2,3,\ldots$
Iterative Deepening Search

Repeat Depth-limited search with L=1,2,3,\ldots

Limit = 1

 TU Dresden, 15th April 2014  PSSAI  slide 39 of 44
Iterative Deepening Search

Repeat Depth-limited search with $L=1, 2, 3, \ldots$

Limit = 2

TU Dresden, 15th April 2014
Iterative Deepening Search

Repeat Depth-limited search with $L=1,2,3,\ldots$
## Properties of Iterative Deepening Search

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>((d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d))</td>
</tr>
<tr>
<td>Space</td>
<td>(O(bd))</td>
</tr>
<tr>
<td>Optimal</td>
<td>Yes, if step cost = 1</td>
</tr>
</tbody>
</table>
Properties of Iterative Deepening Search

Complete  Yes
Time  \((d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d)\)
Space  \(O(bd)\)
Optimal  Yes, if step cost = 1

Number of nodes generated in worst case for \(b = 10\) and \(d = 5\) (solution at far right leaf):

\[
N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \\
N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,100
\]

Hybrid approach that runs BFS until almost all memory is consumed, and then runs IDS from all the nodes in the frontier. In general, IDS is the preferred uninformed search method when the search space is large and the depth of the solution is not known.
References

Zbigniew Michalewicz and David B. Fogel.

Stuart J. Russell and Peter Norvig.