Agenda

1. Introduction
2. Uninformed Search versus Informed Search (Best First Search, A* Search, Heuristics)
3. Local Search, Stochastic Hill Climbing, Simulated Annealing
4. Tabu Search
5. Answer-set Programming (ASP)
6. Constraint Satisfaction (CSP)
7. Structural Decomposition Techniques (Tree/Hypertree Decompositions)
8. Evolutionary Algorithms/ Genetic Algorithms
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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0100</td>
<td>4</td>
<td>1000</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>0101</td>
<td>5</td>
<td>1001</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>0110</td>
<td>6</td>
<td>1010</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>0111</td>
<td>7</td>
<td>1011</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>1001</td>
<td>9</td>
<td>1100</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>1010</td>
<td>10</td>
<td>1101</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>1011</td>
<td>11</td>
<td>1110</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>1111</td>
<td>15</td>
<td></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 \( x_1 = f \) (FALSE), and the second contains all vectors where \( x_1 = t \) (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 gen1_permutation(i)**

1: \( k \leftarrow k + 1 \)
2: \( P[i] \leftarrow k \)
3: if \( k = n \) then
4:   for \( q = 1 \) to \( n \) do
5:     print \( P[q] \)
6:   end for
7: end if
8: for \( q = 1 \) to \( n \) do
9:   if \( P[q] = 0 \) then
10:      gen1_permutation(q)
11:   end if
12: end for
13: \( k \leftarrow k - 1 \)
14: \( P[i] \leftarrow 0 \)
Enumerating the TSP

**Algorithm** gen1_permutation($i$)

1: $k \leftarrow k + 1$
2: $P[i] \leftarrow k$
3: if $k = n$ then
4: for $q = 1$ to $n$ do
5: print $P[q]$
6: end for
7: end if
8: for $q = 1$ to $n$ do
9: if $P[q] = 0$ then
10: gen1_permutation($q$)
11: end if
12: end for
13: $k \leftarrow k - 1$
14: $P[i] \leftarrow 0$

- 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 $n = 3$: $(1\ 2\ 3)$, $(1\ 3\ 2)$, $(2\ 1\ 3)$, $(3\ 1\ 2)$, $(2\ 3\ 1)$, $(3\ 2\ 1)$. 

TU Dresden, 28th April 2015

PSSAI

slide 13 of 84
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.
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
Suppose the SAT formula $\varphi$ contains a clause $(x_1 \lor x_2)$.

TU Dresden, 28th April 2015
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.
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 $L=1, 2, 3, \ldots$
Iterative Deepening Search

Repeat Depth-limited search with L=1,2,3,...
Iterative Deepening Search

Repeat Depth-limited search with $L=1,2,3,\ldots$
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**: \(O(b^d)\)

Optimal**: Yes, if step cost = 1
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.
Agenda

1. Introduction
2. Uninformed Search versus Informed Search (Best First Search, A* Search, Heuristics)
3. Local Search, Stochastic Hill Climbing, Simulated Annealing
4. Tabu Search
5. Answer-set Programming (ASP)
6. Constraint Satisfaction (CSP)
7. Structural Decomposition Techniques (Tree/Hypertree Decompositions)
8. Evolutionary Algorithms/ Genetic Algorithms
Best-First Search

Idea: use an evaluation function for each node
  • estimate of “desirability”
⇒ Expand most desirable unexpanded node

Special cases:
  • Greedy search
  • Dynamic programming
  • A* search
Example: Romania with step costs in km

**Straight-line distance to Bucharest**

- **Arad**: 366
- **Bucharest**: 0
- **Craiova**: 160
- **Dobreta**: 242
- **Eforie**: 161
- **Fagaras**: 178
- **Giurgiu**: 77
- **Hirsova**: 151
- **Iasi**: 226
- **Lugoj**: 244
- **Mehadia**: 241
- **Neamt**: 234
- **Oradea**: 380
- **Pitesti**: 98
- **Rimnicu Vilcea**: 193
- **Sibiu**: 253
- **Timisoara**: 329
- **Urziceni**: 80
- **Vaslui**: 199
- **Zerind**: 374
Greedy Search

- Evaluation function $h(n)$ (heuristic)
  
  $h(n)$ = estimate of cost from $n$ to the closest goal

- E.g., $h_{SLD}(n)$ = straight-line distance from $n$ to Bucharest

- Greedy search expands the node that appears to be closest to goal
Greedy Search Example
Greedy Search Example

- Zerind
- Arad
- Sibiu
- Timisoara

253 329 374
Greedy Search Example

TU Dresden, 28th April 2015

PSSAI
Greedy Search Example

TU Dresden, 28th April 2015

PSSAI
Properties of Greedy Search

**Complete**? No—can get stuck in loops, e.g.,
- Iasi → Neamt → Iasi → Neamt →
- Complete in finite space with repeated-state checking

**Time**? $O(b^m)$, but a good heuristic can give dramatic improvement

**Space**? $O(b^m)$—keeps all nodes in memory

**Optimal**? No
Dynamic Programming

Principle of finding an overall solution by operating on an intermediate point that lies between where you are now and where you want to go.

- Procedure is recursive, each next intermediate point is a function of the points already visited.
- Prototypical problem suitable for dynamic programming has the following properties.
Dynamic Programming

Principle of finding an overall solution by operating on an intermediate point that lies between where you are now and where you want to go.

- Procedure is recursive, each next intermediate point is a function of the points already visited.
- Prototypical problem suitable for dynamic programming has the following properties.

- Can be decomposed into a sequence of decisions made at various stages.
- Each stage has a number of possible states.
- A decision takes you from a state at one stage to some state at the next stage.
- Best sequence of decisions (policy) at any stage is independent of the decisions made at prior stages.
- Well-defined cost for traversing from state to state across stages.
- There is a recursive relationship from choosing the best decisions to make.
Dynamic Programming ctd.

Procedure

- Starting at the goal and working backward to the current state.
- First, determine the best decision at last stage.
- From there, determine the best decision at the next to last stage, presuming we will make the best decision at the last stage.
- And so forth . . .
Dynamic Program for the TSP

\[
L = \begin{bmatrix}
0 & 7 & 12 & 8 & 11 \\
3 & 0 & 10 & 7 & 13 \\
4 & 8 & 0 & 9 & 12 \\
6 & 6 & 9 & 0 & 10 \\
7 & 7 & 11 & 10 & 0 \\
\end{bmatrix}
\]

- Suppose, we start from city 1.
- We split the problem into smaller problems.
- \(g(i, S)\) length of the shortest path from city \(i\) to 1 that passes through each city in \(S\).
- \(g(4, \{5, 2, 3\})\) is the shortest path from city 4 through cities 5, 2 and 3 (in some unspecified order) and then returns to 1.
- \(g(1, V - \{1\})\) is the length of the shortest complete tour.
- In general, we claim that

\[
g(i, S) = \min_{j \in S} \{L(i, j) + g(j, S - \{j\})\}.
\]
Dynamic Program for the TSP ctd.

The problem is to find \( g(1, \{2, 3, 4, 5\}) \).
We start backwards with \( S = \emptyset \).

\[
L = \begin{bmatrix}
0 & 7 & 12 & 8 & 11 \\
3 & 0 & 10 & 7 & 13 \\
4 & 8 & 0 & 9 & 12 \\
6 & 6 & 9 & 0 & 10 \\
7 & 7 & 11 & 10 & 0
\end{bmatrix}
\]

\[
g(2, \emptyset) = L(2, 1) = 3,
g(3, \emptyset) = L(3, 1) = 4,
g(4, \emptyset) = L(4, 1) = 6, \text{ and}
g(5, \emptyset) = L(5, 1) = 7.
\]
Dynamic Program for the TSP ctd.

\[
L = \begin{bmatrix}
0 & 7 & 12 & 8 & 11 \\
3 & 0 & 10 & 7 & 13 \\
4 & 8 & 0 & 9 & 12 \\
6 & 6 & 9 & 0 & 10 \\
7 & 7 & 11 & 10 & 0
\end{bmatrix}
\]

Next iteration, find the solutions to all problems where \(|S| = 1\) (12 sub-problems).

\[
g(2, \{3\}) = L(2, 3) + g(3, \emptyset) = 10 + 4 = 14,
\]
\[
g(2, \{4\}) = L(2, 4) + g(4, \emptyset) = 7 + 6 = 13, \text{ and}
\]
\[
g(2, \{5\}) = L(2, 5) + g(5, \emptyset) = 13 + 7 = 20.
\]
Dynamic Program for the TSP ctd.

For city 3:

\[ g(3, \{2\}) = L(3, 2) + g(2, \emptyset) = 8 + 3 = 11, \]
\[ g(3, \{4\}) = L(3, 4) + g(4, \emptyset) = 9 + 6 = 15, \]
\[ g(3, \{5\}) = L(3, 5) + g(5, \emptyset) = 12 + 7 = 19. \]

For city 4:

\[ g(4, \{2\}) = L(4, 2) + g(2, \emptyset) = 6 + 3 = 9, \]
\[ g(4, \{3\}) = L(4, 3) + g(3, \emptyset) = 9 + 4 = 13, \]
\[ g(4, \{5\}) = L(4, 5) + g(5, \emptyset) = 10 + 7 = 17. \]

For city 5:

\[ g(5, \{2\}) = L(5, 2) + g(2, \emptyset) = 7 + 3 = 10, \]
\[ g(5, \{3\}) = L(5, 3) + g(3, \emptyset) = 11 + 4 = 15, \]
\[ g(5, \{4\}) = L(5, 4) + g(4, \emptyset) = 10 + 6 = 16. \]
Dynamic Program for the TSP ctd.

\[
L = \begin{bmatrix}
0 & 7 & 12 & 8 & 11 \\
3 & 0 & 10 & 7 & 13 \\
4 & 8 & 0 & 9 & 12 \\
6 & 6 & 9 & 0 & 10 \\
7 & 7 & 11 & 10 & 0
\end{bmatrix}
\]

Next iteration, |\(S| = 2\).

\[
g(2, \{3, 4\}) = \min\{L(2, 3) + g(3, \{4\}), L(2, 4) + g(4, \{3\})\} \\
= \min\{10 + 15, 7 + 13\} = \min\{25, 20\} = 20,
\]

\[
g(2, \{3, 5\}) = \min\{L(2, 3) + g(3, \{5\}), L(2, 5) + g(5, \{3\})\} \\
= \min\{10 + 19, 13 + 15\} = \min\{29, 28\} = 28,
\]

\[
g(2, \{4, 5\}) = \min\{L(2, 4) + g(4, \{5\}), L(2, 5) + g(5, \{4\})\} \\
= \min\{7 + 17, 13 + 16\} = \min\{24, 29\} = 24.
\]
Dynamic Program for the TSP ctd.

For city 3:

\[
g(3, \{2, 5\}) = \min\{L(3, 2) + g(2, \{5\}), L(3, 5) + g(5, \{2\})\} \\
= \min\{8 + 20, 12 + 10\} = \min\{28, 22\} = 22,
\]

\[
g(3, \{2, 4\}) = \min\{L(3, 2) + g(2, \{4\}), L(3, 4) + g(4, \{2\})\} \\
= \min\{8 + 13, 9 + 9\} = \min\{21, 18\} = 18,
\]

\[
g(3, \{4, 5\}) = \min\{L(3, 4) + g(4, \{5\}), L(3, 5) + g(5, \{4\})\} \\
= \min\{9 + 17, 12 + 16\} = \min\{26, 28\} = 26.
\]

For city 4:

\[
g(4, \{2, 3\}) = \min\{L(4, 2) + g(2, \{3\}), L(4, 3) + g(3, \{2\})\} \\
= \min\{6 + 14, 9 + 11\} = \min\{20, 20\} = 20,
\]

\[
g(4, \{2, 5\}) = \min\{L(4, 2) + g(2, \{5\}), L(4, 5) + g(5, \{2\})\} \\
= \min\{6 + 20, 10 + 10\} = \min\{26, 20\} = 20,
\]

\[
g(4, \{3, 5\}) = \min\{L(4, 3) + g(3, \{5\}), L(4, 5) + g(5, \{3\})\} \\
= \min\{9 + 19, 10 + 15\} = \min\{28, 25\} = 25.
\]
For city 5:

\[
g(5, \{2, 3\}) = \min\{L(5, 2) + g(2, \{3\}), L(5, 3) + g(3, \{2\})\}
\]
\[= \min\{7 + 14, 11 + 11\} = \min\{21, 22\} = 21,
\]

\[
g(5, \{2, 4\}) = \min\{L(5, 2) + g(2, \{4\}), L(5, 4) + g(4, \{2\})\}
\]
\[= \min\{7 + 13, 10 + 19\} = \min\{20, 29\} = 20,
\]

\[
g(5, \{3, 4\}) = \min\{L(5, 3) + g(3, \{4\}), L(5, 4) + g(4, \{3\})\}
\]
\[= \min\{11 + 15, 10 + 13\} = \min\{26, 23\} = 23.
\]
Dynamic Program for the TSP ctd.

\[
L = \begin{bmatrix}
0 & 7 & 12 & 8 & 11 \\
3 & 0 & 10 & 7 & 13 \\
4 & 8 & 0 & 9 & 12 \\
6 & 6 & 9 & 0 & 10 \\
7 & 7 & 11 & 10 & 0 \\
\end{bmatrix}
\]

Next iteration, \(|S| = 3\).

\[
g(2, \{3, 4, 5\}) = \min\{L(2, 3) + g(3, \{4, 5\}), L(2, 4) + g(4, \{3, 5\}), L(2, 5) + g(5, \{3, 4\})\}
\]
\[
= \min\{10 + 26, 7 + 25, 13 + 23\} = \min\{36, 32, 34\} = 32,
\]
Dynamic Program for the TSP ctd.

Next iteration, \(|S| = 3\).

\[
g(2, \{3, 4, 5\}) = \min\{L(2, 3) + g(3, \{4, 5\}), L(2, 4) + g(4, \{3, 5\}), L(2, 5) + g(5, \{3, 4\})\}
\]
\[
= \min\{10 + 26, 7 + 25, 13 + 23\} = \min\{36, 32, 34\} = 32,
\]
\[
g(3, \{2, 4, 5\}) = \min\{L(3, 2) + g(2, \{4, 5\}), L(3, 4) + g(4, \{2, 5\}), L(3, 5) + g(5, \{2, 4\})\}
\]
\[
= \min\{8 + 24, 9 + 20, 12 + 20\} = \min\{32, 29, 32\} = 29,
\]
\[
g(4, \{2, 3, 5\}) = \min\{L(4, 2) + g(2, \{3, 5\}), L(4, 3) + g(3, \{2, 5\}), L(4, 5) + g(5, \{2, 3\})\}
\]
\[
= \min\{6 + 28, 9 + 22, 10 + 21\} = \min\{34, 31, 31\} = 31.
\]
\[
g(5, \{2, 3, 4\}) = \min\{L(5, 2) + g(2, \{3, 4\}), L(5, 3) + g(3, \{2, 4\}), L(5, 4) + g(4, \{2, 3\})\}
\]
\[
= \min\{7 + 20, 11 + 18, 10 + 20\} = \min\{27, 29, 30\} = 27.
\]
Dynamic Program for the TSP ctd.

\[
L = \begin{bmatrix}
0 & 7 & 12 & 8 & 11 \\
3 & 0 & 10 & 7 & 13 \\
4 & 8 & 0 & 9 & 12 \\
6 & 6 & 9 & 0 & 10 \\
7 & 7 & 11 & 10 & 0 \\
\end{bmatrix}
\]

Last iteration, \(|S| = 4\), original problem:

\[
g(1, \{2, 3, 4, 5\}) = \min \{L(1, 2) + g(2, \{3, 4, 5\}), L(1, 3) + g(3, \{2, 4, 5\}), L(1, 4) + g(4, \{2, 3, 5\}), L(1, 5) + g(5, \{2, 3, 4\})\}
\]

\[
= \min \{7 + 32, 12 + 29, 8 + 31, 11 + 27\} = \min \{39, 41, 39, 38\} = 38.
\]

Shortest tour has length 38.
Which tour is that?
Dynamic Program for the TSP ctd.

Last iteration, \(|S| = 4\), original problem:

\[
g(1, \{2, 3, 4, 5\}) = \min\{L(1, 2) + g(2, \{3, 4, 5\}), L(1, 3) + g(3, \{2, 4, 5\}),
L(1, 4) + g(4, \{2, 3, 5\}), L(1, 5) + g(5, \{2, 3, 4\})\}
= \min\{7 + 32, 12 + 29, 8 + 31, 11 + 27\} = \min\{39, 41, 39, 38\} = 38.
\]

Shortest tour has length 38. Which tour is that?

- **Additional data structure** \(W\) with information on the next city with minimal path.
- \(W(1, \{2, 3, 4, 5\}) = 5\).
- \(W(5, \{2, 3, 4\}) = 2, W(2, \{3, 4\}) = 4, W(4, \{3\}) = 3\),
- last we arrive at city 1.
- Length of this tour is \(11 + 7 + 7 + 9 + 4 = 38\).
Properties of Dynamic Programming

- Computationally intensive: $O(n^2 2^n)$.
- DP algorithms tend to be complicated to understand, because the construction of the program depends on the problem.
- How to formulate sub-problems?
**A* Search**

**Idea:** avoid expanding paths that are already expensive

- Evaluation function $f(n) = g(n) + h(n)$
  - $g(n) = \text{cost so far to reach } n$
  - $h(n) = \text{estimated cost to goal from } n$
  - $f(n) = \text{estimated total cost of path through } n \text{ to goal}$

- $A^*$ search uses an admissible heuristic
  - i.e., $h(n) \leq h^*(n)$ where $h^*(n)$ is the **true** cost from $n$.
  - Also require $h(n) \geq 0$, so $h(G) = 0$ for any goal $G$.

- E.g., $h_{SLD}(n)$ never overestimates the actual road distance

**Theorem:** $A^*$ search is optimal
A* Search Example

Arad
366=0+366
A* Search Example
A* Search Example

TU Dresden, 28th April 2015
A* Search Example

TU Dresden, 28th April 2015

PSSAI
A* Search Example

TU Dresden, 28th April 2015
A* Search Example

TU Dresden, 28th April 2015

PSSAI
Suppose some suboptimal goal $G_2$ has been generated and is in the queue. Let $n$ be an unexpanded node on a shortest path to an optimal goal $G_1$.

$$f(G_2) = g(G_2) \quad \text{since } h(G_2) = 0$$
$$f(G_2) > g(G_1) \quad \text{since } G_2 \text{ is suboptimal}$$
$$\geq f(n) \quad \text{since } h \text{ is admissible}$$

Since $f(G_2) > f(n)$, A* will never select $G_2$ for expansion.
Optimality of A* (more useful)

Lemma: A* expands nodes in order of increasing $f$ value*

- Gradually adds “$f$-contours” of nodes (cf. breadth-first adds layers)
- Contour $i$ has all nodes with $f = f_i$, where $f_i < f_{i+1}$
Properties of A*

**Complete??** Yes, unless there are infinitely many nodes with $f \leq f(G)$

**Time??** Exponential in [relative error in $h \times$ length of soln.]

**Space??** Keeps all nodes in memory

**Optimal??** Yes—cannot expand $f_{i+1}$ until $f_i$ is finished

- $C^*$ - cost of the optimal solution path
- A* expands all nodes with $f(n) < C^*$
- A* expands some nodes with $f(n) = C^*$
- A* expands no nodes with $f(n) > C^*$

TU Dresden, 28th April 2015
Proof of Lemma: Consistency

A heuristic is consistent if

\[ h(n) \leq c(n, a, n') + h(n') \]

If \( h \) is consistent, we have

\[
\begin{align*}
  f(n') &= g(n') + h(n') \\
        &= g(n) + c(n, a, n') + h(n') \\
        &\geq g(n) + h(n) \\
        &= f(n)
\end{align*}
\]

I.e., \( f(n) \) is nondecreasing along any path.
Admissible Heuristics

E.g., for the 8-puzzle:

- \( h_1(n) \) = number of misplaced tiles
- \( h_2(n) \) = total Manhattan distance (i.e., no. of squares from desired location of each tile)

\[
\begin{array}{ccc}
7 & 2 & 4 \\
5 & 6 & \\
8 & 3 & 1 \\
\end{array}
\]

Start State

\[
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & \\
\end{array}
\]

Goal State

\[
h_1(S) = \text{??}
\]
\[
h_2(S) = \text{??}
\]
Admissible Heuristics

E.g., for the 8-puzzle:

- $h_1(n) =$ number of misplaced tiles
- $h_2(n) =$ total Manhattan distance (i.e., no. of squares from desired location of each tile)

\[ h_1(S) = ?? \]
\[ h_2(S) = ?? \]

\[ 6 \]
\[ 4 + 0 + 3 + 3 + 1 + 0 + 2 + 1 = 14 \]
Dominance

If \( h_2(n) \geq h_1(n) \) for all \( n \) (both admissible) then \( h_2 \) dominates \( h_1 \) and is better for search.

Typical search costs:

\[
\begin{align*}
    d = 14 & \quad \text{IDS} = 3,473,941 \text{ nodes} \\
            & \quad A^*(h_1) = 539 \text{ nodes} \\
            & \quad A^*(h_2) = 113 \text{ nodes} \\
    d = 24 & \quad \text{IDS} \approx 54,000,000,000 \text{ nodes} \\
            & \quad A^*(h_1) = 39,135 \text{ nodes} \\
            & \quad A^*(h_2) = 1,641 \text{ nodes}
\end{align*}
\]

Given any admissible heuristics \( h_a, h_b \),

\[
h(n) = \max(h_a(n), h_b(n))
\]

is also admissible and dominates \( h_a, h_b \)
Relaxed problems

- Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem.
- If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution.
- If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution.

Key point: the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem.
Summary

- Heuristic functions estimate costs of shortest paths
- Good heuristics can dramatically reduce search cost
- Greedy best-first search expands lowest $h$:
  - incomplete and not always optimal
- Dynamic programming:
  - complete and optimal
  - time and space consuming
  - how to define the sub-problems?
- A* search expands lowest $g + h$:
  - complete and optimal
  - also optimally efficient
- Admissible heuristics can be derived from exact solution of relaxed problems
References

Zbigniew Michalewicz and David B. Fogel. 

Stuart J. Russell and Peter Norvig. 