UIT2504 Artificial Intelligence

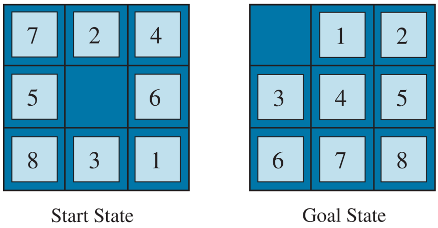
Heuristics Revisited

C. Aravindan
<AravindanC@ssn.edu.in>

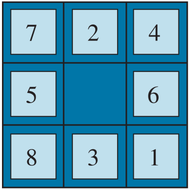
Professor of Computing
SSN College of Engineering

August 21, 2024

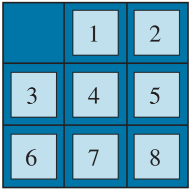
Heuristics — Revisited



Heuristics — Revisited



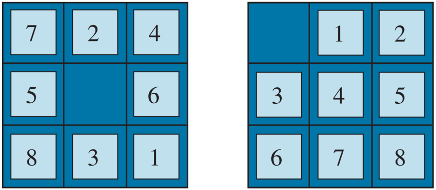
Start State



Goal State

- Even for such a simple problem, the search space is huge!

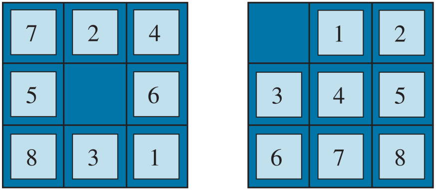
Heuristics — Revisited



Start State Goal State

- Even for such a simple problem, the search space is huge!
- Average depth (22) and average branching factor (3) imply searching about $3^{22} \approx 3.1x10^{10}$ states!!!

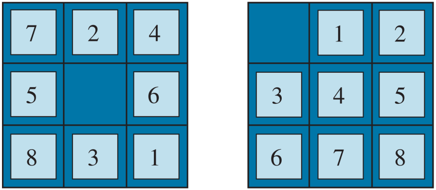
Heuristics — Revisited



Start State Goal State

- Even for such a simple problem, the search space is huge!
- Average depth (22) and average branching factor (3) imply searching about $3^{22} \approx 3.1x10^{10}$ states!!!
- What about higher order sliding puzzle problems?

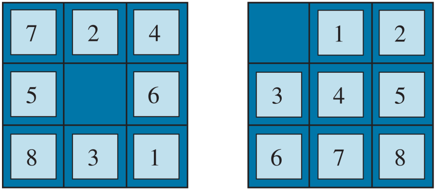
Heuristics — Revisited



Start State   Goal State

- Even for such a simple problem, the search space is huge!
- Average depth (22) and average branching factor (3) imply searching about $3^{22} \approx 3.1x10^{10}$ states!!!
- What about higher order sliding puzzle problems?
- Use of good heuristics can drastically cut down the search space!

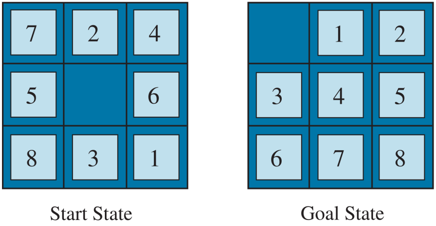
Heuristics — Revisited



Start State   Goal State

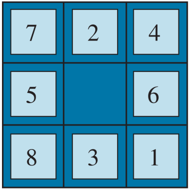
- h_1: Number of misplaced tiles
- h_2: Sum of the (vertical + horizontal) distances of the tiles from their goal positions (total Manhattan distance)
- For the given start state, $h_1 = 8$, and $h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$

Heuristics — Revisited

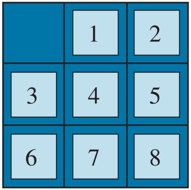


- h_1: Number of misplaced tiles
- h_2: Sum of the (vertical + horizontal) distances of the tiles from their goal positions (total Manhattan distance)
- For the given start state, $h_1 = 8$, and $h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$
- It can be shown that both h_1 and h_2 are admissible

Heuristics — Revisited



Start State



Goal State

- h_1: Number of misplaced tiles
- h_2: Sum of the (vertical + horizontal) distances of the tiles from their goal positions (total Manhattan distance)
- For the given start state, $h_1 = 8$, and $h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$
- It can be shown that both h_1 and h_2 are admissible
- As discussed already, h_2 dominates h_1

Dominant Heuristics

- Does dominance matter?

Dominant Heuristics

- Does dominance matter?
- Performance of a heuristics may be measured in terms of effective branching factor b^*

Dominant Heuristics

- Does dominance matter?
- Performance of a heuristics may be measured in terms of effective branching factor b^*
- If N nodes are generated to find a solution at depth d, then b^* is the branching factor of a uniform tree to generate N nodes with depth d

$$N = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d$$

Dominant Heuristics

- Does dominance matter?
- Performance of a heuristics may be measured in terms of effective branching factor b^*
- If N nodes are generated to find a solution at depth d, then b^* is the branching factor of a uniform tree to generate N nodes with depth d

$$N = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d$$

- Effective branching factor is determined by the relative error in estimation $\epsilon = (h^* - h)/h^*$

Dominant Heuristics

- Does dominance matter?
- Performance of a heuristics may be measured in terms of effective branching factor b^*
- If N nodes are generated to find a solution at depth d, then b^* is the branching factor of a uniform tree to generate N nodes with depth d

$$N = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d$$

- Effective branching factor is determined by the relative error in estimation $\epsilon = (h^* - h)/h^*$
- A^* using h_2 does not expand more nodes than the one using h_1

Dominance matters!

	Search Cost (nodes generated)			Effective Branching Factor		
d	BFS	A*(h_1)	A*(h_2)	BFS	A*(h_1)	A*(h_2)
6	128	24	19	2.01	1.42	1.34
8	368	48	31	1.91	1.40	1.30
10	1033	116	48	1.85	1.43	1.27
12	2672	279	84	1.80	1.45	1.28
14	6783	678	174	1.77	1.47	1.31
16	17270	1683	364	1.74	1.48	1.32
18	41558	4102	751	1.72	1.49	1.34
20	91493	9905	1318	1.69	1.50	1.34
22	175921	22955	2548	1.66	1.50	1.34
24	290082	53039	5733	1.62	1.50	1.36
26	395355	110372	10080	1.58	1.50	1.35
28	463234	202565	22055	1.53	1.49	1.36

Finding heuristics from relaxed problems

- Solution to a simplified relaxed version of a problem may be a good heuristics to solve the original problem

Finding heuristics from relaxed problems

- Solution to a simplified relaxed version of a problem may be a good heuristics to solve the original problem
- For example, for the sliding puzzle problem, if we relax that a tile can move to a non-empty (neighbour) square also, we get h_2

Finding heuristics from relaxed problems

- Solution to a simplified relaxed version of a problem may be a good heuristics to solve the original problem
- For example, for the sliding puzzle problem, if we relax that a tile can move to a non-empty (neighbour) square also, we get h_2
- If we relax further that a tile can jump to any cell, we get h_1

Finding heuristics from relaxed problems

- Solution to a simplified relaxed version of a problem may be a good heuristics to solve the original problem
- For example, for the sliding puzzle problem, if we relax that a tile can move to a non-empty (neighbour) square also, we get h_2
- If we relax further that a tile can jump to any cell, we get h_1
- Exercise: Think of a relaxed problem where a tile can move from X to Y, if Y is blank.

Finding heuristics from relaxed problems

- Solution to a simplified relaxed version of a problem may be a good heuristics to solve the original problem
- For example, for the sliding puzzle problem, if we relax that a tile can move to a non-empty (neighbour) square also, we get h_2
- If we relax further that a tile can jump to any cell, we get h_1
- Exercise: Think of a relaxed problem where a tile can move from X to Y, if Y is blank. Solution to this relaxed problem gives yet another heuristics!

Finding heuristics from relaxed problems

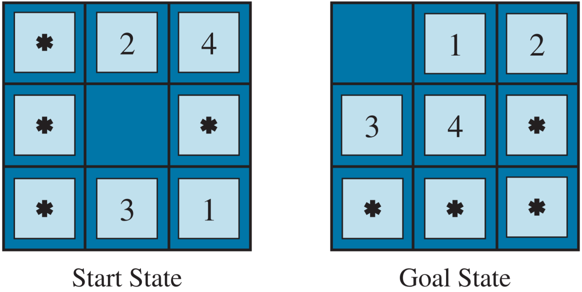
- Solution to a simplified relaxed version of a problem may be a good heuristics to solve the original problem
- For example, for the sliding puzzle problem, if we relax that a tile can move to a non-empty (neighbour) square also, we get h_2
- If we relax further that a tile can jump to any cell, we get h_1
- Exercise: Think of a relaxed problem where a tile can move from X to Y, if Y is blank. Solution to this relaxed problem gives yet another heuristics!
- When there are many heuristics with no clarity on which is better (dominant), we can think of a composite heuristics $h(n) = max\,\{h_1(n), h_2(n), \cdots, h_k(n)\}$

Generating heuristics from subproblems

- Admissible heuristics can be derived from the solution cost of a subproblem of a given problem

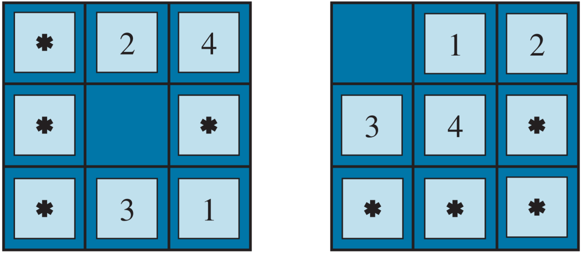
Generating heuristics from subproblems

- Admissible heuristics can be derived from the solution cost of a subproblem of a given problem



Generating heuristics from subproblems

- Admissible heuristics can be derived from the solution cost of a subproblem of a given problem



Start State Goal State

- Solution to this subproblem turns out to be a better heuristics than the Manhattan distance for all the nodes that fit this pattern

Pattern databases

- We may create pattern databases to store the costs of solving every possible subproblem instance matching this pattern (there will be 15,120 instances)

Pattern databases

- We may create pattern databases to store the costs of solving every possible subproblem instance matching this pattern (there will be 15,120 instances)
- The heuristics $h_{DB}(n)$ can be computed by looking up n in the pattern database

Pattern databases

- We may create pattern databases to store the costs of solving every possible subproblem instance matching this pattern (there will be 15,120 instances)
- The heuristics $h_{DB}(n)$ can be computed by looking up n in the pattern database
- It is easy to construct such a database, as we can compute the actual cost of a pattern by working backwards from the goal states!

Pattern databases

- We may create pattern databases to store the costs of solving every possible subproblem instance matching this pattern (there will be 15,120 instances)
- The heuristics $h_{DB}(n)$ can be computed by looking up n in the pattern database
- It is easy to construct such a database, as we can compute the actual cost of a pattern by working backwards from the goal states!
- Similarly, other patterns such as $5-6-7-8$ may be considered

Pattern databases

- We may create pattern databases to store the costs of solving every possible subproblem instance matching this pattern (there will be 15,120 instances)
- The heuristics $h_{DB}(n)$ can be computed by looking up n in the pattern database
- It is easy to construct such a database, as we can compute the actual cost of a pattern by working backwards from the goal states!
- Similarly, other patterns such as $5-6-7-8$ may be considered
- Heuristic values from different pattern databases may be easily composed (using max function) to form a stronger heuristics

Pattern Databases

- Is it possible to add the heuristic values from disjoint pattern databases? (for example from $1-2-3-4$ and $5-6-7-8$ databases)

Pattern Databases

- Is it possible to add the heuristic values from disjoint pattern databases? (for example from $1-2-3-4$ and $5-6-7-8$ databases)
- In general, the answer is no, since the resulting heuristics may not be admissible

Pattern Databases

- Is it possible to add the heuristic values from disjoint pattern databases? (for example from $1-2-3-4$ and $5-6-7-8$ databases)
- In general, the answer is no, since the resulting heuristics may not be admissible
- However, if we count only the moves involving the tiles $1, 2, 3, 4$ in the $1-2-3-4$ and so on, then summation is possible

Generating heuristics from landmarks

- Apps, such as Google Maps, are able to find "optimal" paths between two nodes very fast from a map of innumerable nodes

Generating heuristics from landmarks

- Apps, such as Google Maps, are able to find "optimal" paths between two nodes very fast from a map of innumerable nodes
- The basic idea for faster execution is to precompute some optimal paths in the graph

Generating heuristics from landmarks

- Apps, such as Google Maps, are able to find “optimal” paths between two nodes very fast from a map of innumerable nodes
- The basic idea for faster execution is to precompute some optimal paths in the graph
- Precomputation may be very costly, but need to be done only once (hopefully!)

Generating heuristics from landmarks

- Apps, such as Google Maps, are able to find "optimal" paths between two nodes very fast from a map of innumerable nodes
- The basic idea for faster execution is to precompute some optimal paths in the graph
- Precomputation may be very costly, but need to be done only once (hopefully!)
- If the map is not "huge", we can even try all-pairs shortest paths!

Generating heuristics from landmarks

- Apps, such as Google Maps, are able to find "optimal" paths between two nodes very fast from a map of innumerable nodes
- The basic idea for faster execution is to precompute some optimal paths in the graph
- Precomputation may be very costly, but need to be done only once (hopefully!)
- If the map is not "huge", we can even try all-pairs shortest paths!
- However, in general, it may not be feasible, and a better approach will be to select a few landmark points among the vertices, and for each landmark L, compute $C^*(v, L)$ for all the vertices

Generating heuristics from landmarks

- Apps, such as Google Maps, are able to find "optimal" paths between two nodes very fast from a map of innumerable nodes
- The basic idea for faster execution is to precompute some optimal paths in the graph
- Precomputation may be very costly, but need to be done only once (hopefully!)
- If the map is not "huge", we can even try all-pairs shortest paths!
- However, in general, it may not be feasible, and a better approach will be to select a few landmark points among the vertices, and for each landmark L, compute $C^*(v, L)$ for all the vertices
- In general, we may need $C^*(L, v)$ as well

Generating heuristics from landmarks

- Apps, such as Google Maps, are able to find "optimal" paths between two nodes very fast from a map of innumerable nodes
- The basic idea for faster execution is to precompute some optimal paths in the graph
- Precomputation may be very costly, but need to be done only once (hopefully!)
- If the map is not "huge", we can even try all-pairs shortest paths!
- However, in general, it may not be feasible, and a better approach will be to select a few landmark points among the vertices, and for each landmark L, compute $C^*(v, L)$ for all the vertices
- In general, we may need $C^*(L, v)$ as well
- An efficient heuristic: $h_L(n) = \min_{L \in Landmarks} C^*(n, L) + C^*(L, goal)$

Generating heuristics from landmarks

- Apps, such as Google Maps, are able to find "optimal" paths between two nodes very fast from a map of innumerable nodes
- The basic idea for faster execution is to precompute some optimal paths in the graph
- Precomputation may be very costly, but need to be done only once (hopefully!)
- If the map is not "huge", we can even try all-pairs shortest paths!
- However, in general, it may not be feasible, and a better approach will be to select a few landmark points among the vertices, and for each landmark L, compute $C^*(v, L)$ for all the vertices
- In general, we may need $C^*(L, v)$ as well
- An efficient heuristic: $h_L(n) = \min_{L \in Landmarks} C^*(n, L) + C^*(L, goal)$
- This will be the best heuristics, if L is along the optimal path to goal. Otherwise, it may be an overestimate.

Differential heuristic

- If we choose our landmarks carefully, an efficient and admissible heuristic can be used:

$$h_{DH}(n) = \max_{L \in Landmarks} |C^*(n, L) - C^*(goal, L)|$$

Differential heuristic

- If we choose our landmarks carefully, an efficient and admissible heuristic can be used:

$$h_{DH}(n) = \max_{L \in Landmarks} |C^*(n, L) - C^*(goal, L)|$$

- This is called as a Differential Heuristic

Differential heuristic

- If we choose our landmarks carefully, an efficient and admissible heuristic can be used:

$$h_{DH}(n) = \max_{L \in Landmarks} |C^*(n, L) - C^*(goal, L)|$$

- This is called as a Differential Heuristic
- How do we select the landmark points?

Differential heuristic

- If we choose our landmarks carefully, an efficient and admissible heuristic can be used:

$$h_{DH}(n) = \max_{L \in Landmarks} |C^*(n, L) - C^*(goal, L)|$$

- This is called as a Differential Heuristic
- How do we select the landmark points?
- Greedy approach: Select a vertex at random, then add a point which is farthest from that, and continue to add at each iteration, a point which is farther away from all the previous points

Differential heuristic

- If we choose our landmarks carefully, an efficient and admissible heuristic can be used:

$$h_{DH}(n) = \max_{L \in Landmarks} |C^*(n, L) - C^*(goal, L)|$$

- This is called as a Differential Heuristic
- How do we select the landmark points?
- Greedy approach: Select a vertex at random, then add a point which is farthest from that, and continue to add at each iteration, a point which is farther away from all the previous points
- Landmarks may be selected from the history of user searches

Differential heuristic

- If we choose our landmarks carefully, an efficient and admissible heuristic can be used:

$$h_{DH}(n) = \max_{L \in Landmarks} |C^*(n, L) - C^*(goal, L)|$$

- This is called as a Differential Heuristic
- How do we select the landmark points?
- Greedy approach: Select a vertex at random, then add a point which is farthest from that, and continue to add at each iteration, a point which is farther away from all the previous points
- Landmarks may be selected from the history of user searches
- For DH, it is better if the landmarks are spread around the perimeter of the graph — arrange k pie-shaped wedges around the centroid and select the farthest vertex in each wedge

Learning to search

- Memory may be an issue in finding an optimal solution

Learning to search

- Memory may be an issue in finding an optimal solution
- None of A^* and its variants is guaranteed to solve *any* complex problem with given memory and time

Learning to search

- Memory may be an issue in finding an optimal solution
- None of A^* and its variants is guaranteed to solve *any* complex problem with given memory and time
- Reinforcement learning techniques may be used to learn some meta-heuristics to guarantee completeness, but may be at the cost of optimality

Learning to search

- Memory may be an issue in finding an optimal solution
- None of A^* and its variants is guaranteed to solve *any* complex problem with given memory and time
- Reinforcement learning techniques may be used to learn some meta-heuristics to guarantee completeness, but may be at the cost of optimality
- Machine learning techniques may be used to learn heuristics from experience

Learning to search

- Memory may be an issue in finding an optimal solution
- None of A^* and its variants is guaranteed to solve *any* complex problem with given memory and time
- Reinforcement learning techniques may be used to learn some meta-heuristics to guarantee completeness, but may be at the cost of optimality
- Machine learning techniques may be used to learn heuristics from experience
- A neural network may be trained to map a state to a heuristic value

Learning Heuristics

- By solving several instances of a problem, we will have a good collection of actual cost of reaching goal from different states

Learning Heuristics

- By solving several instances of a problem, we will have a good collection of actual cost of reaching goal from different states
- Heuristics function may now be learnt from these examples

Learning Heuristics

- By solving several instances of a problem, we will have a good collection of actual cost of reaching goal from different states
- Heuristics function may now be learnt from these examples — inductive learning

Learning Heuristics

- By solving several instances of a problem, we will have a good collection of actual cost of reaching goal from different states
- Heuristics function may now be learnt from these examples — inductive learning
- Typically, the heuristics function to be learnt will be expressed in terms of weighted combination of features

$$h(n) = w_1 f_1 + w_2 f_2 + \cdots + w_n f_n$$

Learning Heuristics

- By solving several instances of a problem, we will have a good collection of actual cost of reaching goal from different states
- Heuristics function may now be learnt from these examples — inductive learning
- Typically, the heuristics function to be learnt will be expressed in terms of weighted combination of features

$$h(n) = w_1 f_1 + w_2 f_2 + \cdots + w_n f_n$$

- Examples of features: "Number of misplaced tiles", "number of pairs of adjacent tiles that are not adjacent in the goal state"

Learning Heuristics

- By solving several instances of a problem, we will have a good collection of actual cost of reaching goal from different states
- Heuristics function may now be learnt from these examples — inductive learning
- Typically, the heuristics function to be learnt will be expressed in terms of weighted combination of features

$$h(n) = w_1 f_1 + w_2 f_2 + \cdots + w_n f_n$$

- Examples of features: "Number of misplaced tiles", "number of pairs of adjacent tiles that are not adjacent in the goal state"
- Several learning algorithms are available to learn these weights from examples

Questions?