ARTIFICIAL INTELLIGENCE

The Very Idea

Vasant G. Honavar
Dorothy Foehr Huck and J. Lloyd Huck Chair in Biomedical Data Sciences and Artificial Intelligence
Professor of Data Sciences, Informatics, Computer Science, Bioinformatics & Genomics and Neuroscience
Director, Artificial Intelligence Research Laboratory
Director, Center for Artificial Intelligence Foundations and Scientific Applications
Associate Director, Institute for Computational and Data Sciences
Pennsylvania State University

vhonavar@psu.edu
http://faculty.ist.psu.edu/vhonavar
http://ailab.ist.psu.edu

Problem Solving Agents

Pizza Run: The Scenario

- Hungry Joe is relaxing at home on a Saturday afternoon
- He starts salivating about pizza for lunch
- He wants to be eating a pizza
- Time for a pizza run!

Pizza Run: The problem to be solved

- Current State: Hungry, sitting on the couch, at home
- Desired Goal State: Sitting at the table, eating pizza
- Solution: A plan or a sequence of actions that you can execute to go from the Current State to the desired Goal State

Pizza Run: The Plan

- Get off the couch, get dressed
- Grab wallet and car keys
- Leave apartment, walk to car
- Drive to nearest pizza place
- Order, pay, and pick up pizza
- Return home with pizza
- Sit down, open the box, enjoy!

How could Hungry Joe solve the problem?

- Given the start state and the goal state, and a set of actions at his disposal
  - Hungry Joe had to find a suitable sequence of actions
    - for a successful pizza run
    - that if correctly executed, would take him from the start state to the goal state

Generalizing the pizza run – state space search

- Many problems share the same structure
- The agent is given
  - a start state
  - a goal state
  - a set of actions, each with its own
    - Pre-conditions
      - conditions that must hold before an action is executed
    - Post-conditions
      - Conditions that hold after an action is executed
- Task: Find a sequence of actions that, if successfully executed, will take the agent from the start state to the desired goal state

State-space problem representation

A state space search problem is specified by

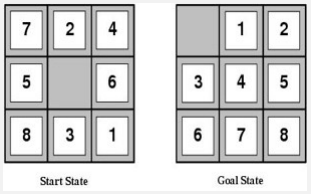
- *S is* the set of possible start states
- *O* is the set of actions
  - Partial functions that map a state into another
  - Partial because not every action may apply in every state
- *G* the set of goal states
  - *G* may be explicitly enumerated or implicitly specified using a goal predicate *goal* (*g*) = *True* iff *g* is a goal

Solution to a state space search problem is a sequence of actions leading from the start state $s \in S$ to a goal $g \in G$

State-space problem formulation

- Design a state representation that
  - Captures the relevant aspects of the world
  - Abstracts away unimportant details
- Formulate the goals
  - Specification of the solution
    - Explicit (enumeration of goal states)
    - Implicit (goal predicate)
- Formulate the actions
  - Preconditions
    - The conditions that need to hold before an action can be executed
  - Post-conditions
    - The conditions that are guaranteed to hold after an action is (successfully) executed

Example: 8-puzzle

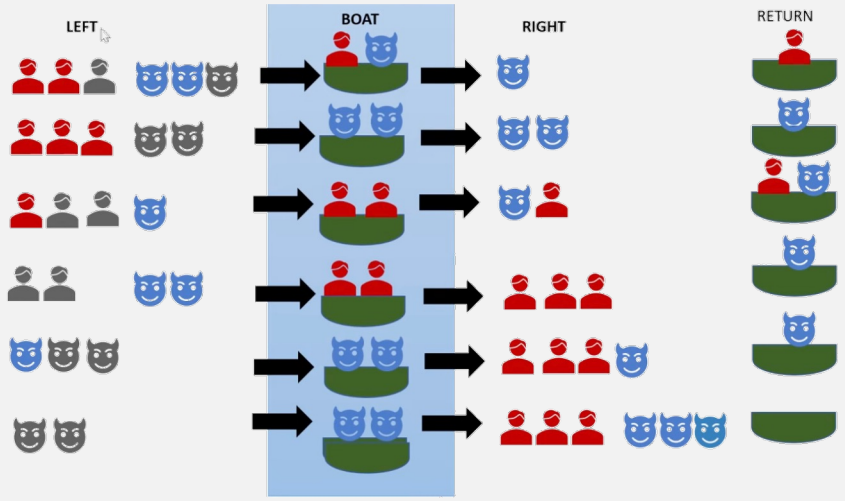


- States: Location of each tile on the board
- Initial state: Any state can be initial
- Actions:
  - *Left* – swap the blank tile with the tile to its left
  - *Right* – swap the blank tile with tile to its right
  - *Up* – swap the blank tile with tile above it
  - *Down* -- swap the blank tile with the tile below it
- Goal test: Check whether goal configuration is reached
- Solution: A sequence of actions

Example: Missionaries and Cannibals

- Initial state: 3 missionaries, 3 cannibals, and the boat on the left bank of the river
- Goal: all on the right bank
- Constraints:
  - The boat which can carry at most 2 people at a time
  - If missionaries are outnumbered by cannibals, the cannibals will eat the missionaries
- States: The positions of missionaries, cannibals, and the boat on either side of the river
- Actions: Movement of the boat with its occupants from one side of the river to the other, subject to the constraints
- Solution: A sequence of boat trips across the river complete with their passenger lists

Example: Missionaries and Cannibals



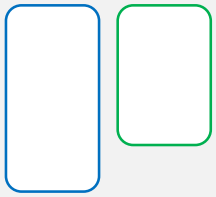
Water jug problem

- You have two jugs – A, B – with capacities of 4 liters and 3 liters
- Neither has measurement markings
- Actions:
  - Fill a jug completely
  - Empty a jug completely
  - Pour water from one jug to another until it is full or the first one is empty
- Goal: Measure exactly 2 liters of water

Water jug problem

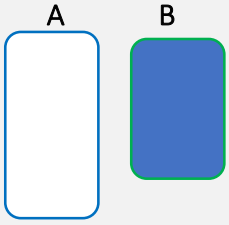
- You have two jugs – A, B – with capacities of 4 liters and 3 liters
- Neither has measurement markings
- Actions:
  - Fill a jug completely
  - Empty a jug completely
  - Pour water from one jug to another until it is full or the first one is empty
- Goal: Measure exactly 2 liters of water
- Representation
  - States: (x, y), where x = water in Jug A, y = water in Jug B
  - Initial State: (0, 0) (both empty)
  - Goal State: (x, 2) (Jug B has 2 liters, A can have any amount)

Some states of the Water Jug Problem



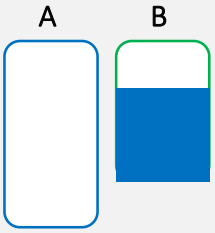
(0,0)

Initial State



(0,3)

One state directly reachable from the Initial State



(0,2)

A possible Goal state

A possible solution to the Water Jug problem

- You have two jugs – A, B – with capacities of 4 liters and 3 liters
- Actions:
  - Fill a jug completely
  - Empty a jug completely
  - Pour water from one jug to another until it is full or the first is empty
- Goal: Measure exactly 2 liters of water

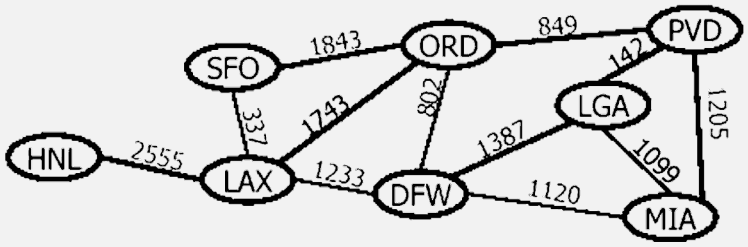
A possible solution

(0, 0) → Fill Jug B → (0, 3)→ Pour Jug B into Jug A → (3, 0)→ Fill Jug B → (3, 3) → Pour Jug B into Jug A → (4, 2) → Empty Jug A → (0, 2)

Example: Getting around in USA

- Working in SF
  - There is a graduation to attend in MIA
- Goal
  - To be in MIA
- Problem formulation
  - States: Various cities
  - Actions: Fly between cities connected by air
- Solution
  - Shortest route between SF and MIA

Shortest path problem



Problem formulation in the observable, deterministic world

- A problem is defined by:
  - An initial state, e.g. SFO
  - Actions
    - SFO → ORD
    - SFO → LAX
    - …..
  - Goal test
    - Current state = MIA?
- Initial state + successor function defines a state space
- A solution is a sequence of actions from the initial to goal state

Basic State Space Search Problem

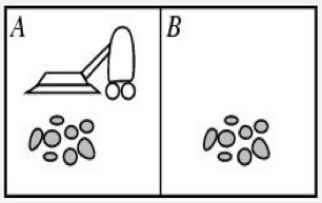
A state space search problem is specified by a 3-tuple

(*S, O, G*) where

- *S* is a set of possible start states
- *O* is the set of actions (operators)
  - Partial functions that map a state into another
- *G* the set of goal states
  - *G* may be explicitly enumerated or implicitly specified using a goal test *goal* (*g*) = *True* iff *g* is a goal state

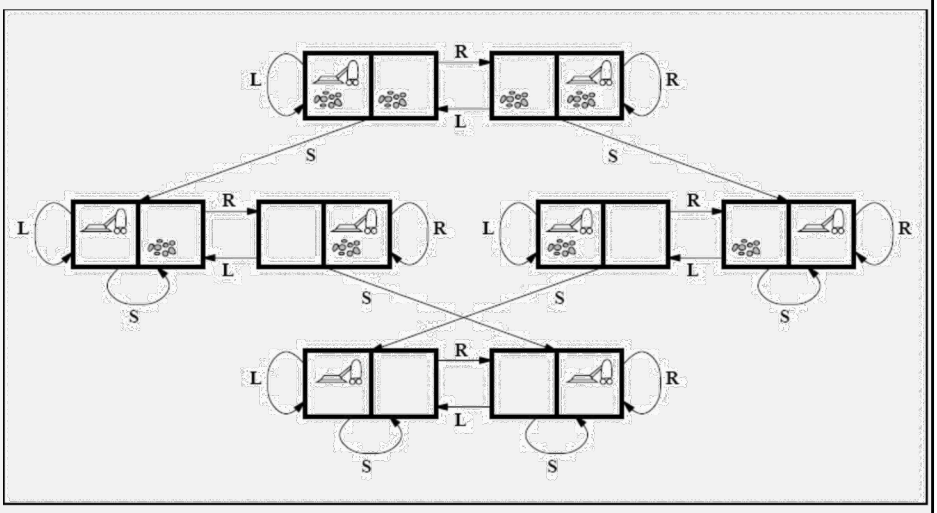
Solution to a state space search problem is a sequence of actions leading from the start state *s* to a goal $g \in G$

Example: vacuum world

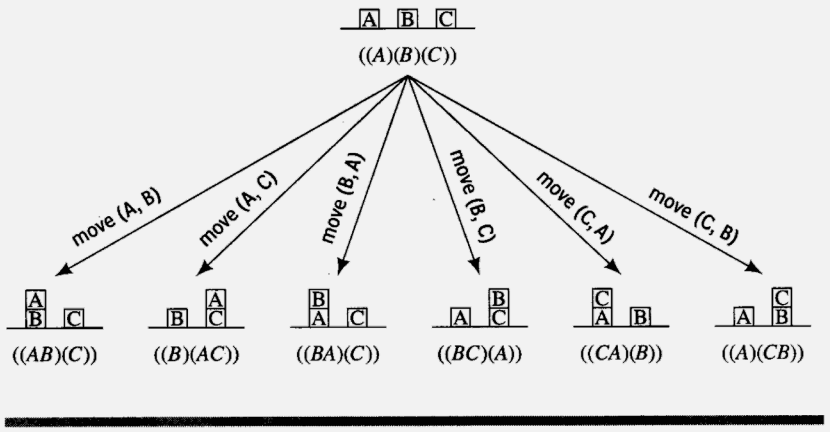


- States? two locations with or without dirt, with or without the vacuum cleaner: 2 x 2^2=8 states.
- Initial state? Any state can be initial
- Actions? {*Left*, *Right*, *Cleanup*}
- Goal test? Check whether both locations are clean.

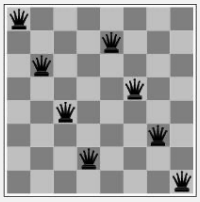
Example: Complete State Space of the Vacuum World



Blocks World



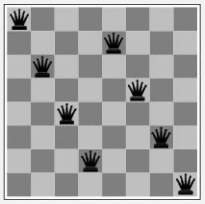
Example: 8-queens problem



- State: Any arrangement of Queens on the board
- Initial State: Empty board
- Actions: Placing a queen in an empty square
- Constraints:
  - Queens attack each other if they share
    - A row
    - A column
    - A diagonal
- Goal:
  - All 8 queens on the board with no queen attacking any other

State space representation: 8-queens problem

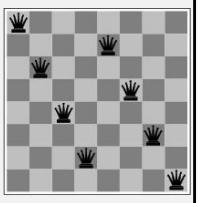
Problem formulation 1



- State: Any arrangement of 0 to 8 queens on the board
- 64 squares, 8 queens
  - $(64)(63)(62)(61)..(57) \approx 3 \times 10^{14} \approx 1.2681 \times 2^{47}$ states!
- Can we do better?

State space representation: 8-queens problem

Problem Formulation 2



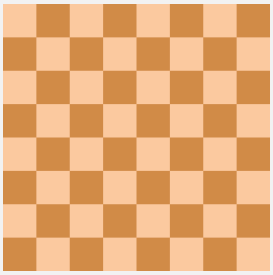
- State - any arrangement of 0 to 8 queens on the board
- 8 rows – need to specify the column in which a queen is placed in each row
  - Rows are variables which can take values in {1, 2, 3, 4, 5, 6, 7, 8}
  - The first row can take 8 values, second 7, third 6, ..
  - $(8)(7)(6)(5)(4)(3)(2)(1) \approx 1.231 \times 2^{15}$ states!
  - Absorbed the `no two queens can share a row' constraint into the representation!
  - Much better than formulation 1 with 1.2681×2^{47} states!
  - Can we do better?

Example: 8-queens problem

Problem formulation 3

- State: Any arrangement of 0 to 8 queens on the board
- States: ($0 \leq n \leq 8$) queens on the board, one per column in the n leftmost columns with no queen attacking another
- Actions: Add a queen to the leftmost empty column so as not to attack the other queens already on the board
- Backtrack if a dead-end is reached
- Number of states = 2057

Representation matters!



8 Queens Problem

- Number of queens placed on the board = k
- The corresponding number of states N_k
- Initial state – empty board $(k = 0)$, $N_0 = 1$
- Next step – 1 queen on the board $(k = 1)$, $N_1 = 8$ (any row would do)
- Next step – 2 queens on the board $(k = 2)$

O	X						
X	X						
X							
X							
X							
X							
X							
X							

- With the first queen in column 1,
  - Row 1, Column 1, and diagonal are blocked
  - Leaving 6 possible row placements for the second queen in column 2
- The situation is similar first queen is placed in row 8

8 Queens Problem

- Number of queens placed on the board = k
- The corresponding number of states N_k
- Initial state – empty board $(k = 0)$, $N_0 = 1$
- Next step – 1 queen on the board $(k = 1)$, $N_1 = 8$ (any row would do)
- Next step – 2 queens on the board $(k = 2)$

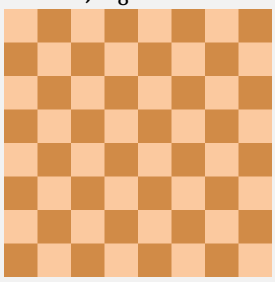
X	**X**						
O	X						
X	**X**						
X							
X							
X							
X							
X							

- With the first queen in row 2,
  - Row 2, Column 1, and 2 diagonals are blocked
  - Leaving 5 possible row placements for the second queen in column3
- The situation is similar first queen is placed in rows 3, 4, ..7

8 Queens Problem

- Number of queens placed on the board = k
- The corresponding number of states N_k
- Initial state – empty board $(k = 0)$, $N_0 = 1$
- Next step – 1 queen on the board $(k = 1)$, $N_1 = 8$ (any row would do)
- Next step – 2 queens on the board $(k = 2)$, $N_2 = 2{\times}6 + 5{\times}6 = 42$
- Proceeding similarly, we can show that
  $N_3 = 140, N_4 = 344, N_5 = 568, N_6 = 550, N_7 = 312, N_8 = 92$
- Hence, the total number of states

$$\sum_{k=0}^{8} N_k = 2057$$



Representation matters!

Basic State Space Search Problem

A state space search problem is specified by

- *S is* the set of possible start states
- *O* is the set of actions (operators)
  - Partial functions that map a state into another
- *G* the set of goal states
  - *G* may be explicitly enumerated or implicitly specified using a goal predicate *goal* (*g*) = *True* iff *g* is a goal

Solution to a state space search problem is a sequence of actions leading from the start state *s* to a goal $g \in G$

Finding solution – State space search

Let L be a list of nodes yet to be expanded

1. Let *L* = list of partial paths to be extended
2. If *L* is *empty,* return *failure*
   else pick a partial path *p* from *L* (which one?)
3. If *p ends* in a goal node,
   a. return path from *s* to *p* and stop.
   b. Otherwise
      i. *Delete p* from *L*
      ii. Expand *p*: Add to *L* all of *p's* 1-step extensions (where on the list?)
4. Return to 2.

Finding an optimal solution

- All actions may not be equally expensive
- Suppose we have a cost for each step
- c (q,o,r) = cost of applying action o in state q to reach state r
- Path cost is typically assumed to be the sum of costs of operator applications along the path
- An optimal solution is one with the lowest cost path from the specified start state s to a goal $g \in G$

Basic Search strategies

A search strategy specifies a particular order of node expansion
Search strategies are evaluated in terms of:

- Completeness:
  - Does it always find a solution if one exists?
- Admissibility
  - Does it always find an optimal solution?
- Complexity
  - Time Complexity: Number of partial paths explored
  - Space Complexity: Memory needed to store *L* during search
- Optimality
  - Optimal in its use of space, time, or both

Uninformed (blind) search strategies

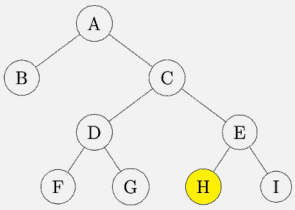
- Use only information available in problem definition
- Blind search strategies:
  - Breadth-first search
  - Depth-first search
  - Iterative deepening search
  - Bidirectional search

Some basic search algorithms

- Note 1
  - To illustrate the algorithm, we show the whole state space
  - The problem solving agent does not see the whole state space
  - It starts in the initial state and must discover the rest of the state space through exploration
  - Different algorithms differ in terms how – in which order – they explore the states
- Note 2
  - We use “expand a node” and “extend a (partial) path” interchangeably

Breadth first search

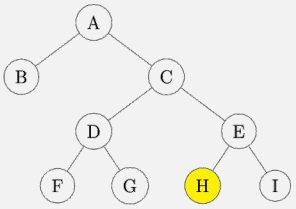
- Breadth-first search starts with the start state as the root of the search tree
- Expand the start state, then expand its successors, then their successors …
  - at each step testing whether the corresponding state is a goal state,
  - until one of the nodes generated passes the goal test or we reach a dead end.
  - At any step if a node tested is not a goal node and has no successors, it is dropped from the list of partial paths.



Breadth first search

We explore the state space level by level

Suppose only H satisfies the goal test



A
AB, AC
ACD, ACE
ACDF, ACDG, ACEH, ACEI
ACDG ACEH ACEI
ACEH ACEI

If the maximum branching factor b is finite, BFS is guaranteed to find a solution if one exists

Memory – exponential in the depth d of the tree

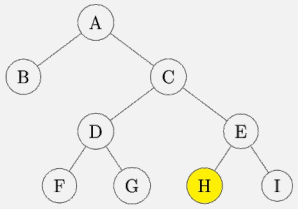
$b^0 + b^1 + \cdots + b^d$

Depth first search

We explore the state space depth first

Suppose only H satisfies the goal test

A
AB AC
AC
ACD ACE
ACDF, ACDG, ACE
ACDG ACE
ACE
ACEH ACEI



- DFS is guaranteed to find a solution if one exists only if the search space is finite and branching factor is finite
- Can fail to terminate if the search space is infinite
- Memory is linear in the depth of the tree 1+(b-1)d

BFS and DFS compared

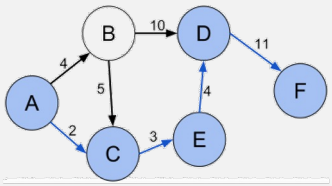
- Advantage of breadth first search
  - Guaranteed to find a solution at finite depth d if one exists, assuming that the branching factor b is finite
- Disadvantage of breadth first search
  - Space complexity is of the order of b^d
- Advantage of depth first search
  - Space complexity is linear in the depth – order of bd
- Disadvantage of depth first search
  - May not terminate with a solution if the tree is infinite, although the solution is of finite depth
- Can we get the best of both?
  - A state space search algorithm that is guaranteed to find a solution if there is one at a finite depth, using memory that is linear in depth of the solution?

Iterative deepening search

- Do depth first searches with depth limited to 0, 1, 2, …
- If there is a solution at depth *d*, it will be found with DFS with depth cutoff *d*
- Are we increasing work?
  - Yes, but by a negligible amount because most of the work is done at depth d because $b^d \gg b^{d-1} \cdots \gg b^0$
  - IDS is guaranteed to find a solution if there is one at a finite depth (assuming branching factor is finite)
- What about space used?
  - At each depth cutoff k where $0 \leq k \leq d$, $1 + k(b-1)$

Finding optimal solutions: Branch and Bound Search

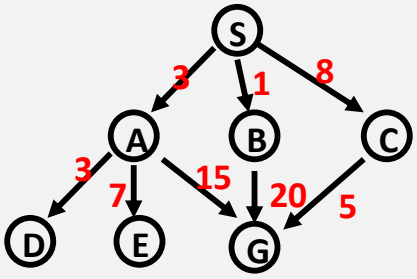
1. A (0)
2. AC (2), AB (4)
3. AB(4), ACE(5)
4. ACE(5), ABC(9), ABD(14)
5. ACED(9), ABC(9), ABD(14)
6. ABC(9), ABD(14), ACEDF(20)
7. ABCE(12), ABD(14), ACEDF(20)
8. ABD(14), ABCED(16), ACEDF(20)
9. ABCED(16), ACEDF(20), ABDF(25)
10. ACEDF(20), ABDF(25), ABCEDF(27)



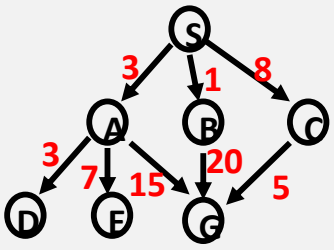
- Now the first path on the list takes us from A to F.
- Because the list is sorted in increasing order of cost, the cost of ACEDF is $\leq$ the costs of all other paths on the list
- Hence, it must be the optimal (cheapest) path from A to F

Exercise

Suppose G is the goal node
Use DFS, BFS and Branch and Bound Search
Which of them is guaranteed to find the optimal solution?



Depth-First Search

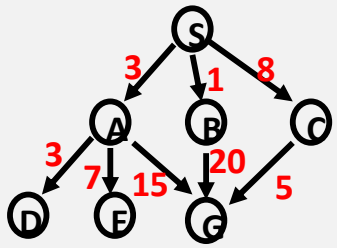


Expanded node	Paths list
	{ S^0 }
S^0	{ SA^3 SB^1 SC^8 }
A^3	{ SAD^6 SAE^{10} SAG^{18} SB^1 SC^8 }
D^6	{ SAE^{10} SAG^{18} SB^1 SCG^{13} }
E^{10}	{ SAG^{18} SB^1 SC^8 }
G^{18}	{ SB^1 SC^8 }

Solution path found is S A G, cost 18

Number of nodes expanded (including goal node) = 5

Breadth-First Search

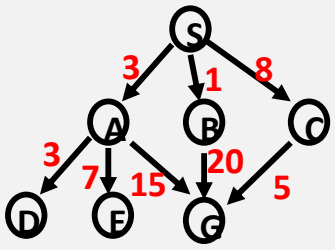


Expanded node	Partial paths list
	{ S^0 }
S^0	{ SA^3 SB^1 SC^8 }
A^3	{ SAB^1 SAC^8 SAD^6 SAE^{10} SAG^{18} }
B^1	{ SAC^8 SAD^6 SAE^{10} SAG^{18} SBG^{21} SCG^{13} }
C^8	{ SAD^6 SAE^{10} SAG^{18} SBG^{21} SCG^{13} }
D^6	{SAE^{10} SAG^{18} SBG^{21} SCG^{13} }
E^{10}	{ SAG^{18} SBG^{21} SCG^{13} }
G^{18}	{ SAG^{21} SCG^{13} }

Solution path found is S A G , cost 18

Number of nodes expanded (including goal node) = 7

Branch and Bound Search



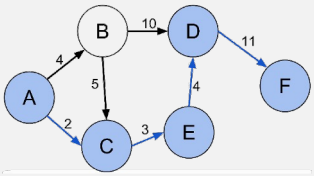
Expanded node	Paths list
	{ S^0 }
S^0	{SB^1 SA^3 SC^8 }
B^1	{SA^3 SC^8 SBG^{21}}
A^3	{ SAD^6 SC^8 SAE^{10} SAG^{18} SBG^{21}}
D^6	{SC^8 SAE^{10} SAG^{18} SBG^{21}}
C^8	{SAE^{10} SCG^{13} SAG^{18} SBG^{21} }
E^8	{ SCG^{13} SAG^{18} SBG^{21} }

Solution path found is S A G, cost 13

Number of nodes expanded (including goal node) = 7

Informed search

- Uninformed search runs out of time, space, or both quickly
- Solution – exploit problem-specific information to guide search
- Problem-specific information is called heuristic
- Perfect heuristic would guide search directly along the cheapest path from the start state to a goal state
  - No need to search!
- In practice, we have imperfect but useful heuristics



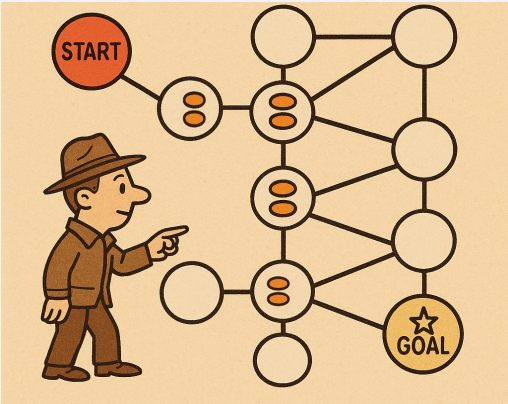
A heuristic function

- Heuristic [dictionary]:
  - "A rule of thumb, simplification, or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood."

Heuristics, those rules of thumb,
Often scorned, as sloppy, dumb,
Yet slowly commonsense become!
- Judea Pearl, in *Heuristics*

Informed search

- Informed search uses heuristics to guide search
  - How to use heuristics to guide search?
  - How to design effective heuristics?



Best first search

- Best-first search
  - Maintain a list of partial paths sorted in increasing order of $f(n)$ which assesses the desirability of the path from the start state to the goal state through n
  - Lower the value of $f(n)$, the more desirable the path to goal that passes through n
  - Choose the path to extend according to $f(n)$ from the front of the list

How to construct f?

- Suppose we could estimate the cost of the cheapest solution obtainable by expanding each node that could be chosen
- A heuristic function $h(n)$ estimates the cost of completing a partial path $(s..n)$ to obtain a solution
  - $h(n)$ is the estimated cost of the sub-path $(n, \ldots g)$ of $(s, \ldots n, .. g)$ where g is a goal state

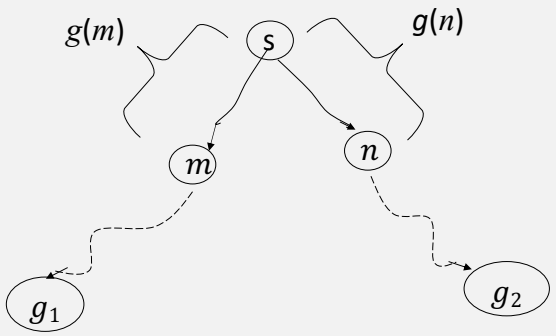
- $h(n)$ maps each state n to a non-negative real number
- In general, $h(n) \geq 0$
- $h(n) = 0$ when n is a goal state

How to construct f?

- Branch-and-bound search
  - $f(n) = g(n)$ the cost of the cheapest path from start node to n
- Greedy best-first search
  - $f(n) = h(n)$ the cost of the path with the cheapest estimated cost from n to a goal
- A* search
  - $f(n)$ is the cost of the path with the cheapest estimated cost to the goal through n
  - $f(n) = g(n) + h(n)$

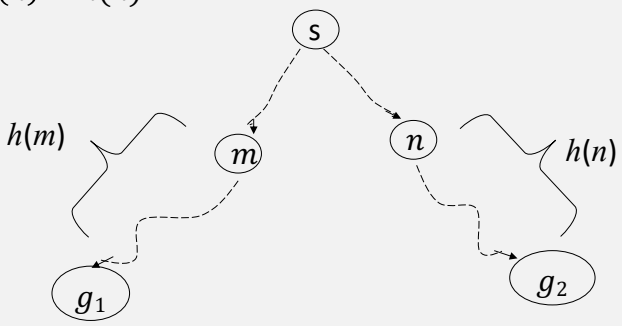
Branch-and-bound search

- Branch and bound search uses the actual cost of the partial paths to order them
- $g(n)$ = actual cost of the cheapest path from n to a goal state
- $f(n) = g(n)$ the cost of the cheapest path from start node to n



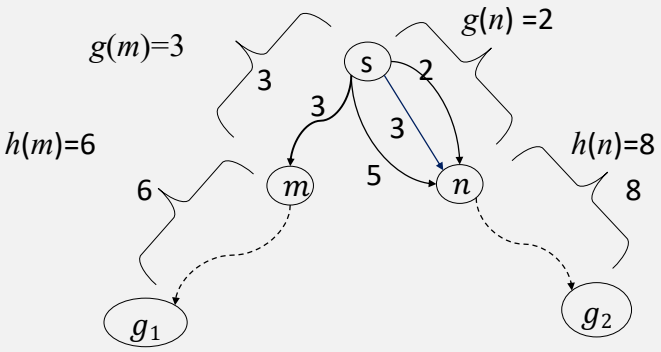
Greedy best-first search

- $h(n)$ = estimated cost of the cheapest path from n to a goal state
- $h(g_1) = h(g_2) = 0$
- $f(n) = h(n)$

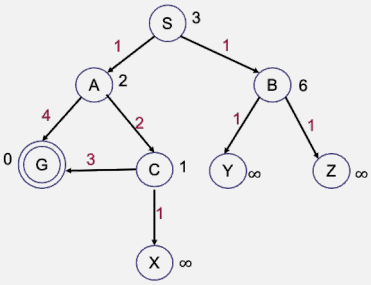


A* search

$$f(n) = g(n) + h(n)$$

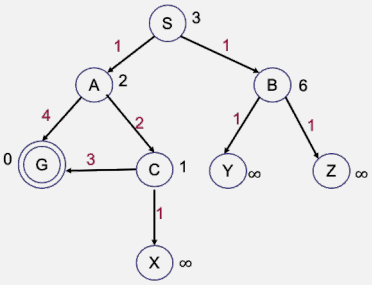


A* search with $f(n) = h(n) + g(n)$



- Numbers next to each state denote the corresponding $h(n)$ values
  - $h(A) = 2, h(B) = 6)$
- Numbers on the arrows denote the cost of the corresponding actions or moves in the state space
  - $Cost(A \rightarrow C) = 2, Cost(S \rightarrow A) = 1$ …

A* search with $f(n) = h(n) + g(n)$



$$(S, 3)$$

$$(SA, 1+2), (SB, 1+6)$$

$$(SAC, 3+1), (SAG, 5+0), (SB, 1+6)$$

$$(SAG, 5), (SACG, 6), (SB, 7)(SACX, \infty)$$

Optimal path SAG with cost of 5 found!

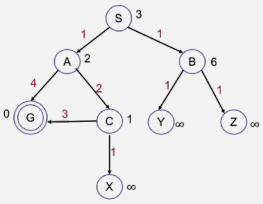
What did we do?

- We maintained a list of partial paths ordered by their f values
- Whenever the path at the front of the list ends in a goal state, we have the optimal path from the start state to the goal state
- Same as branch and bound search
  - Except we use $f(n) = h(n) + g(n)$ instead of $g(n)$ to order the list of partial paths

A* search is a kind of best-first search

- Best-first search
  - Maintain a list of partial paths sorted in increasing order of $f(n)$ which assesses the desirability of the path from the start state to the goal state through n
  - Lower the value of $f(n)$, the more desirable the path to goal that passes through n
  - Choose the path to extend according to $f(n)$ from the front of the list
  - Special cases
    - Branch-and-bound search: $f(n) = g(n)$
    - Greedy search: $f(n) = h(n)$
    - A* search: $f(n) = g(n) + h(n)$

Admissible heuristics



- Let $h^*(n)$ be the actual cost of the optimal path from n to a goal node
- If there is no path from n to goal, $h(n) = \infty$
- The heuristic function $h(n)$ is said to be admissible if for all nodes n, $0 \leq h(n) \leq h^*(n)$

- Is $h(n)$ admissible?
- Cost of the optimal path from S to G = 5. $h(S) = 3 < 5$
- Cost of the optimal path from A to G = 4. $h(A) = 2 < 4$
- Cost of the optimal path from each state to the goal G is less than or equal to the value of the heuristic for the corresponding state
- $h(n)$ shown is admissible

Why do we care if $h(n)$ admissible?

- A* is guaranteed to find an optimal solution from a start state to a goal state whenever
  - $h(n)$ is admissible
  - The cost of each action or move is strictly positive

Admissible heuristic: Example 1

Current state n Goal state

- States: Arrangements of the tiles 1-8 and blank tile on a 3×3 board
- Goal state: a target configuration, typically, with the tiles arranged in a specific order
- Actions: Exchange blank tile with its north, east, west, or south neighbor

- $h_1(n) =$ the number of misplaced numbered tiles relative to goal
- For the state shown, $h_1(n) = 8$
- Is h_1 admissible?
  - Yes!
  - We need at least as many moves as the number of misplaced tiles
  - h_1 underestimates the cost of the optimal sequence of moves from any state to the goal state

Admissible heuristic: Example 1

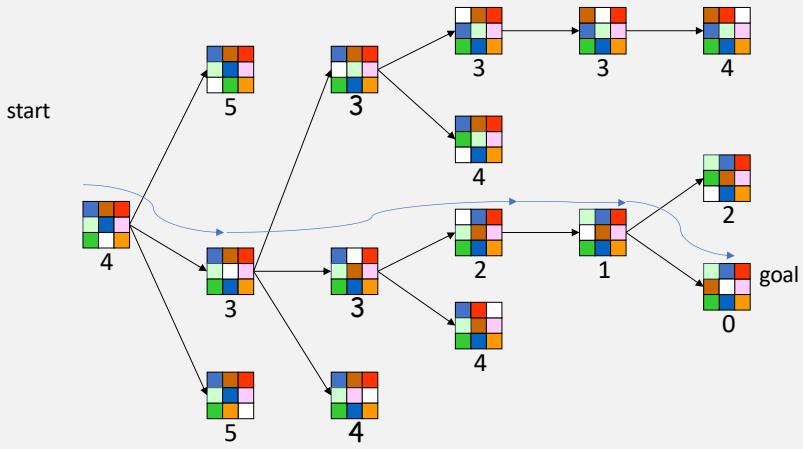
Current state n Goal state

- States: Arrangements of the tiles 1-8 and blank tile on a 3×3 board
- Goal state: a target configuration, typically, with the tiles arranged in a specific order
- Actions: Exchange blank tile with its north, east, west, or south neighbor

- $h_2(n)$ = the sum of the distances of the numbered tiles from their desired positions (Manhattan distance)
  - $h_2(n) = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$ for the state shown
- Is h_2 admissible?
  - Yes!
  - h_2 underestimates the cost of the optimal sequence of moves from any state to the goal state

8-Puzzle

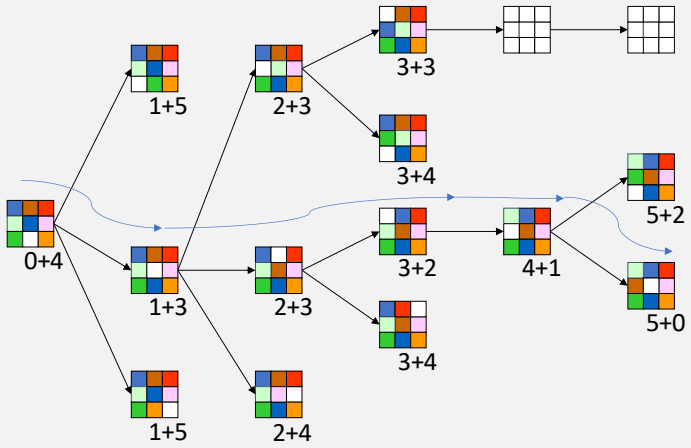
$f(n) = h(n)$ = number of misplaced tiles relative to the goal



The white tile is the empty tile

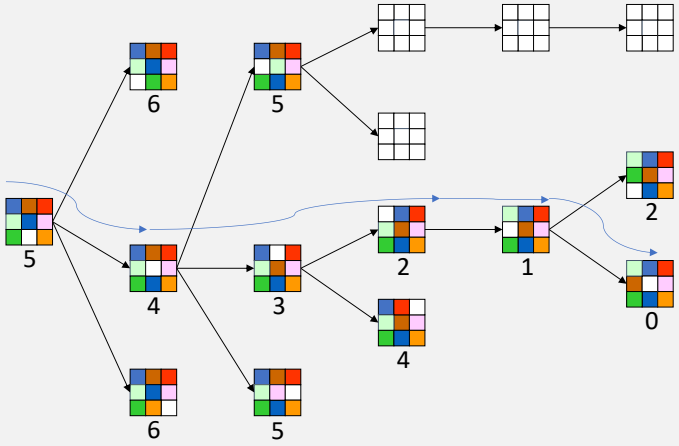
8-Puzzle

$f(n) = g(n) + h(n)$ where the cost of each move = 1
$h(n)$ = number of misplaced numbered tiles relative to the goal



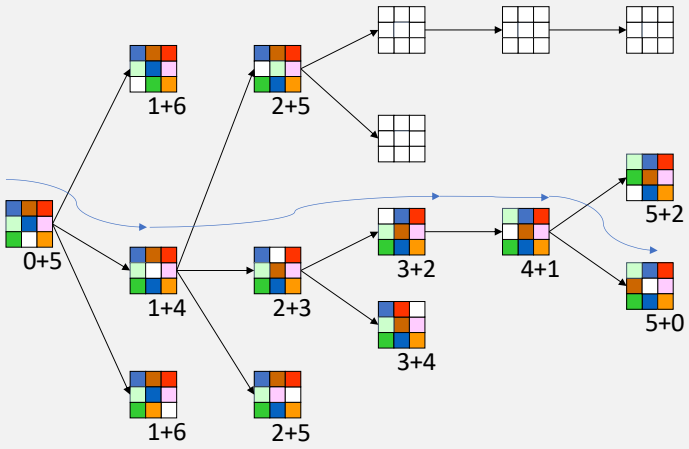
8-Puzzle

$f(n) = h(n) =$ sum of distances of colored tiles to their goals



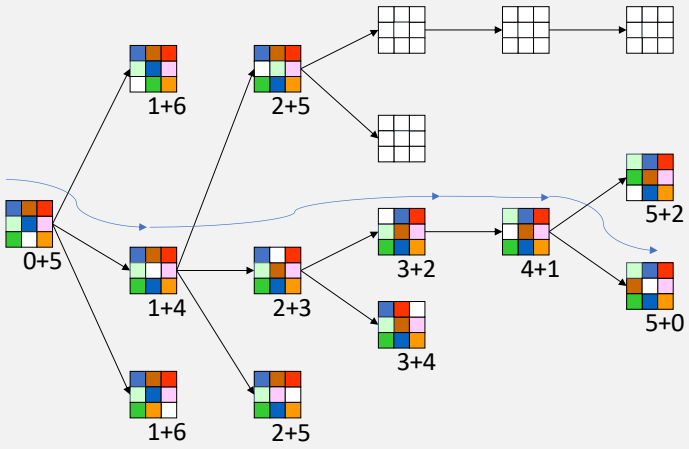
8-Puzzle

$f(n) = g(n) + h(n)$ where
$h(n)$ = sum of city block distances of numbered tiles to their goals



Some observations

- A* with either $f_1(n) = h_1(n) + g(n)$ and $f_2(n) = h_2(n) + g(n)$ yields an optimal solution



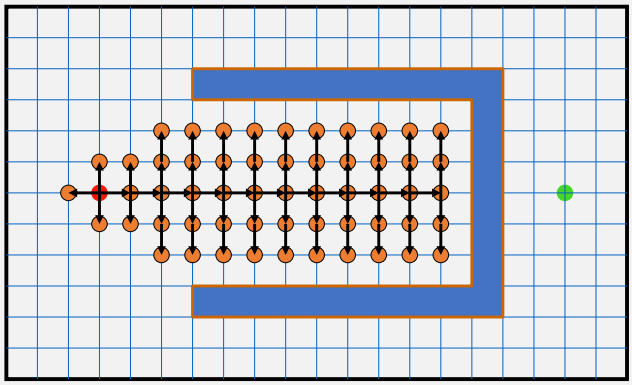
Some observations

- A* using $f_2(n) = h_2(n) + g(n)$ does less work in finding an optimal solution
- A* using $f_2(n) = h_2(n) + g(n)$ explores only a subset of the search space explored using $f_1(n) = h_1(n) + g(n)$
- Why?
- h_2 is more informative than h_1!
- That is, $h_2(n)$ is less of an underestimate the cost of the path through n to a goal node than $h_1(n)$
- The more informative the heuristic, the less work we have to do to find an optimal solution!

Why do we need A* search?

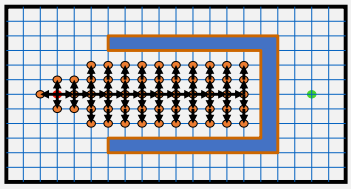
- Greedy search
  - List of partial paths ordered by $f(n) = h(n)$
- Branch-and-bound search
  - List of partial paths ordered by $f(n) = g(n)$
- A* search
  - Branch-and-bound search where the partial paths are ordered according to increasing values of $f(n) = g(n) + h(n)$
- Why do we need A* search?

Problem with best-first search with $f(N) = h(N)$



Suppose $f(N) = h(N)$ = straight distance to the goal
Moves allowed are along the grid, wall is impenetrable

Problem with best-first search with $f(n) = h(n)$



- Relying on $h(n)$ alone is like chasing a mirage in looking for an oasis in a desert
- Each move gives the illusion of taking you closer to the goal
- Better approach: $f(n) = h(n) + g(n)$
- $h(n)$ is the optimist; $g(n)$ is the realist
- $f(n) = g(n)$ will guide you to an optimal solution
- $f(n) = h(n) + g(n)$ will guide you to an optimal solution faster – provided $h(n)$ is a good heuristic

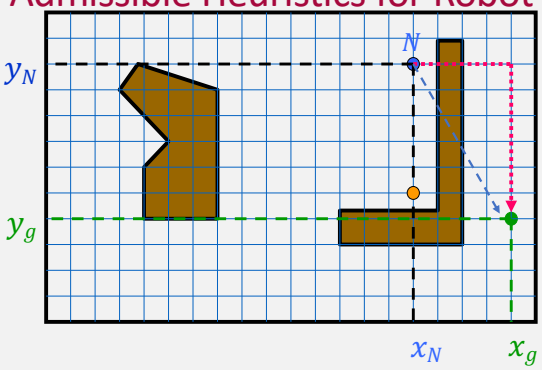
A* works best with admissible heuristics

- Let $h^*(n)$ be the actual cost of the optimal path from n to a goal state
- The heuristic function $h(n)$ is admissible if for all states n, $0 \leq h(n) \leq h^*(n)$

- A* is guaranteed to find an optimal solution from a start state to a goal state whenever
  - $h(n)$ is admissible
  - The cost of each action or move is strictly positive

- How can we design admissible heuristics?

Admissible Heuristics for Robot Navigation



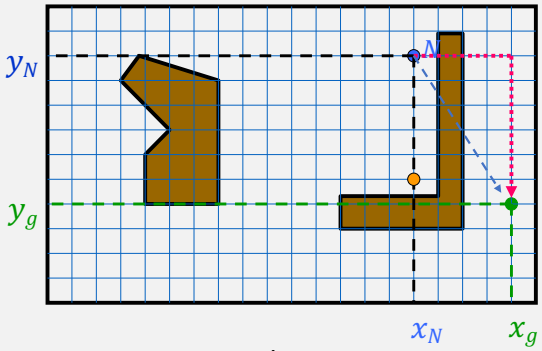
Euclidian distance

$$h_1(N) = \sqrt{(x_N - x_g)^2 + (y_N - y_g)^2}$$

Cost of one horizontal/vertical step = 1

Is this heuristic admissible?
Yes

Admissible Heuristics for Robot Navigation



Manhattan distance

$$h_1(N) = |x_N - x_g| + |y_N - y_g|$$

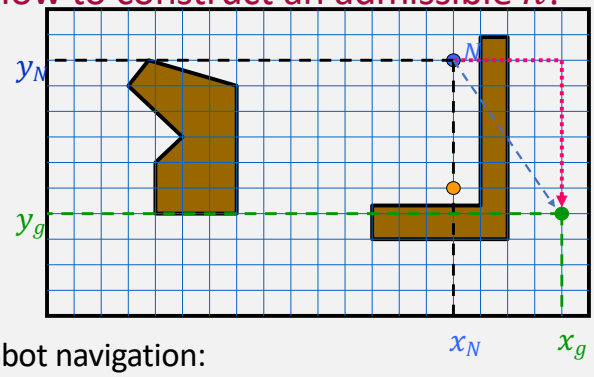
Cost of one horizontal/vertical step = 1

Is this heuristic admissible?
Yes, if diagonal moves are not allowed

How to construct an admissible h?

- An admissible heuristic can be seen as the cost of an optimal solution to a relaxed and hence easier version of the original problem
- How do you get a relaxed version of a problem?
  - Remove one or more problem constraints

How to construct an admissible h?



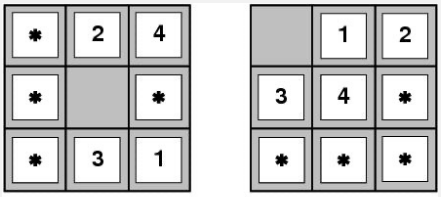
- In robot navigation:
  - The Manhattan distance corresponds to removing the obstacles
  - The Euclidean distance corresponds to removing both the obstacles and the constraint that the robot moves on a grid

Inventing admissible heuristics: Relaxation

- Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem that is easy to solve (without search):
  - Relaxed 8-puzzle for h_1 : a tile can be exchanged with any other tile on the board
  - Relaxed 8-puzzle for h_2 : a tile can move to any adjacent square.
- The optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem
- Heuristic function based on exact cost of the solution of the relaxed problem is guaranteed to be consistent

Inventing admissible heuristics: sub-problems

- Admissible heuristics can be derived from the solution cost of a sub problem of a given problem
- This cost is a lower bound on the cost of the real problem
- Pattern databases store the exact solution to for every possible sub problem instance.
  - The complete heuristic is constructed using the patterns in the DB



A* search

$f(n) = g(n) + h(n)$ where $0 \leq h(n) \leq h^*(n)$

1. Generate 1-hop paths from the start state to its neighbors and order them by their f values
2. If the first path on the list ends in the destination, you have the optimal solution.
3. If not, extend the first path on the list by one step, and update the list, while keeping the list of partial paths sorted according to their f values
4. Go back to step 2 and repeat.

Note that we keep multiple paths to a state if a state is visited more than once

A* search: Additional notes

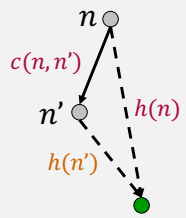
- A* can result in the same state visited multiple times through different paths
- We can leave the paths on the list sorted by their f values
- Or we can drop a more expensive path to a state when a cheaper path is found
- Neither of these approaches is ideal
- Fortunately, for a large family of admissible heuristics
  - the so-called consistent heuristics
  - there is a much more efficient way to handle revisited states

Consistent Heuristic

- We say that an admissible heuristic h is consistent (or monotone) if for each node n and each child n' of n:

$$h(n) \leq c(n, n') + h(n')$$

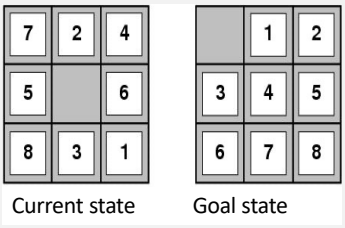
$$h(n) - h(n') \leq c(n, n')$$



(triangle inequality)

- Admissible heuristic underestimates the overall cost of the optimal path from each state to the goal state
- Consistent heuristic underestimates the cost of each of the moves
- Every consistent heuristic is an admissible heuristic
- If $h(n)$ is consistent, whenever a state is expanded, we are guaranteed to have an optimal path to it -- no need to revisit

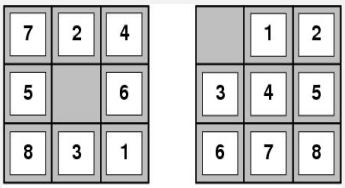
8-Puzzle Heuristics



Current state Goal state

- $h_1(n)$ = number of misplaced tiles
- Is $h_1(n)$ consistent?

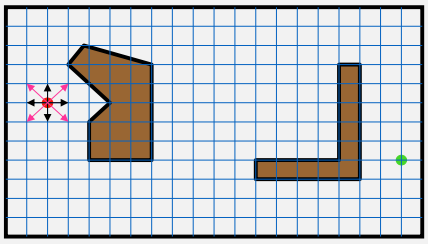
8-Puzzle Heuristics



Current state Goal state

- $h_2(n)$ = sum of the (Manhattan) distances of every tile to its goal position
- Is $h_2(n)$ consistent?

Robot Navigation Heuristics



Cost of one horizontal/vertical step = 1
Cost of one diagonal step = $\sqrt{2}$

Is $h_1(N) = \sqrt{(x_N - x_g)^2 + (y_N - y_g)^2}$ consistent?

Summary: A* with an admissible heuristic

- A heuristic $h(n)$ is admissible if for every state n, $h(n) \leq h^*(n)$, where $h^*(n)$is the true cost of cheapest path to the goal state from n.
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
- If $h(n)$ is admissible, A* is guaranteed to terminate with an optimal solution (provided each arc cost is no smaller than some $\delta > 0$)
- A* is optimal among the family of algorithms that use additive cost functions – finds optimal solution with minimal effort
- What happens when the heuristic overestimates the cost of the optimal solution by an amount, say no greater than ϵ?
  - The solution found is suboptimal by no more than ϵ.

Summary: A* with a consistent heuristic

- A heuristic $h(n)$ is admissible if for every state n, $h(n) \leq h^*(n)$, where $h^*(n)$is the true cost of cheapest path to the goal state from n.
- An admissible heuristic h is consistent (or monotone) if for each state n and each child n' of n: $h(n) \leq c(n, n') + h(n')$
- If $h(n)$ is consistent, A* is guaranteed to terminate with an optimal solution (provided each arc cost is bounded from below by a positive constant δ)
- Furthermore, when A* expands a path $(s \ldots n)$, A* has already found an optimal path from s to the state represented by n.

The more informative the heuristic, the better

- $h(n) = 0$ for all n is consistent, but totally uninformative
- A* with $h(n) = 0$ for all n reduces to branch and bound search
- A* with $h(n) = h^*(n)$ for all n (the perfect heuristic) is as informed as any search algorithm can ever be, so A* proceeds directly along the optimal path from start state to a goal state.
- Given two consistent heuristics $h_1(n)$ and $h_2(n)$, we say that h_2 is more informative than h_1 if $h_2(n) > h_1(n)$ for all n
- If h_2 is more informative than h_1 and A_1* is A* using h_1 and and A_2* is A* using h_2
  - whenever a solution exists, all the nodes expanded by A_2*, except for some nodes such that $f_1(n) = f_2(n) =$ cost of optimal solution are also expanded by A_1*

How good is a heuristic?

- Effective branching factor b^*
  - N = the number of nodes generated by a heuristic search algorithm (e.g., A*)
  - The effective branching factor of search = the branching factor of a tree of depth d needs to have in order to contain $N + 1$ nodes.

    $$N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$$
- Provides a good guide to the heuristic's overall usefulness
- b^* = 1 for A* search with a perfect heuristic

Calculation of effective branching factor

N=20, d=4

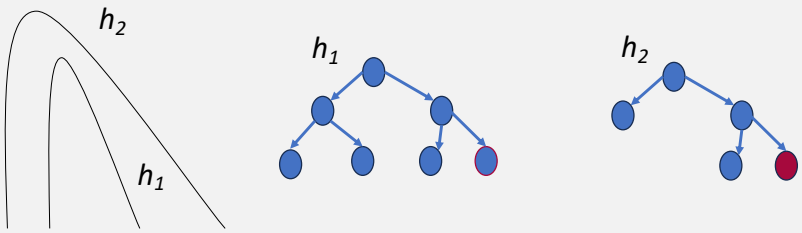
$$N+1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d = \frac{\left[(b^*)^{d+1} - 1\right]}{(b^* - 1)}$$

$$Solve\ for\ b^*: 21 = 1 + b^* + \left(b^*\right)^2 + \left(b^*\right)^3 + \left(b^*\right)^4$$

$$b^* \approx 1.5$$

Dominance

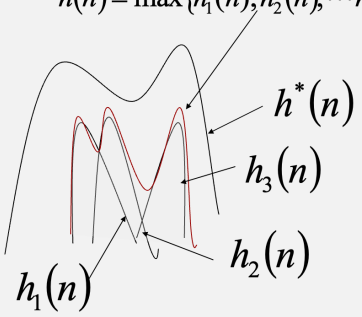
- Given two admissible heuristics h_1 and h_2 such that $h_2(n) \geq h_1(n)$ for all n then h_2 dominates h_1 then h_2 is more informative than h_1
- If h_2 is more informative than h_1 then every partial path that is expanded by A* using h_2 is necessarily expanded by A* using h_1



Can we combine heuristics to get a better heuristic?

- It is often difficult to find a single very good heuristic
- But it is often easy to find a few reasonable heuristics
- When the individual heuristics are admissible or consistent, there is a simple way to combine them to get a heuristic that is at least as good of any of th

$$h(n) = \max\{h_1(n), h_2(n), \cdots h_m(n)\}$$



Memory-bounded heuristic search

- Some solutions to A*'s space problems
  - Iterative-deepening A* (IDA*)
    - Cutoff information is the f-cost $(g+h)$ instead of depth
  - (simple) Memory-bounded A* ((S)MA*)
    - Drop the worst-leaf node when memory is full
  - …

(Simple) Memory-bounded A*

- Use all available memory.
  - i.e. expand best leafs until available memory is full
  - When full, SMA* drops worst leaf node (highest *f*-value) breaking ties arbitrarily
  - Backup the *f*-value of the forgotten node to its parent
- SMA* is complete if solution is reachable, finds optimal solution if optimal solution is reachable within the available memory bound

Problem solving through problem decomposition

Problem decomposition

- Problem
  - Solving an integral
- Sub-problems
  - Easier integrals to solve
- Actions or operators
  - Rules of integral calculus and algebra
- Primitive problems
  - Problems whose solutions can be looked up or computed by executing a known procedure

Example

- Problem
  - solving an integral
- Sub-problems
  - easier integrals to solve
- Operators
  - rules of integral calculus and algebra
- Primitive problems
  - problems whose solutions can be looked up or computed by executing a known procedure

$$\int (2t + sec^2t\ sin^2t + 7\ )dt$$

AND

$$\int 2t\ dt + \int sec^2t\ sin^2t\ dt + \int 7\ dt$$

OR

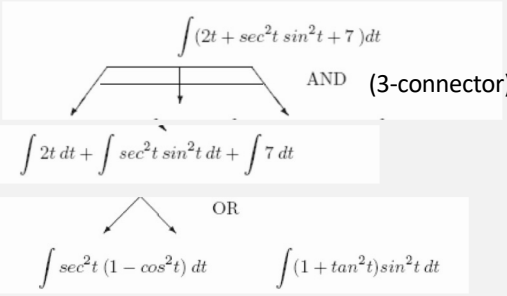
$$\int sec^2t\ (1 - cos^2t)\ dt \qquad \int (1 + tan^2t) sin^2t\ dt$$

Problem reduction representation (PRR)

- A PRR problem is specified by a 3-tuple (*G, O, P*)
  - *G* is a problem to be solved
  - *O* is a set of operators for decomposing problems into sub-problems through AND or OR decompositions
  - *P* is a set of primitive problems with known solutions
- Solution
  - An AND decomposition is solved when each of the sub-problems is solved
  - An OR decomposition is solved when at least one of the sub-problems is solved
  - A problem is unsolvable if it is neither a primitive problem nor can it be further decomposed
- PRR is a generalization of the state space representation (why?)

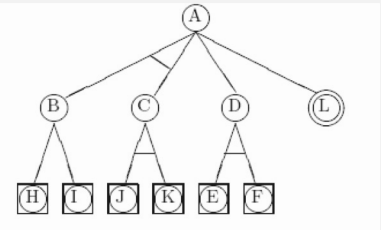
Problem reduction representation

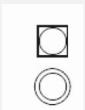
- Solving a problem in PRR reduces to searching an AND-OR graph
- Nodes correspond to problems
- Connectors correspond to actions
- Connectors correspond to AND or OR decompositions
- Connectors of arity *k* are called *k*-connectors



Example

Problem

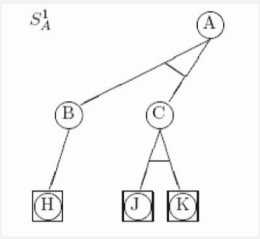


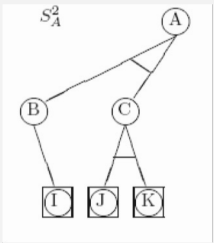


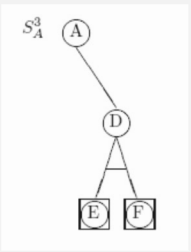
Primitive problem

Unsolvable problem

3 solutions







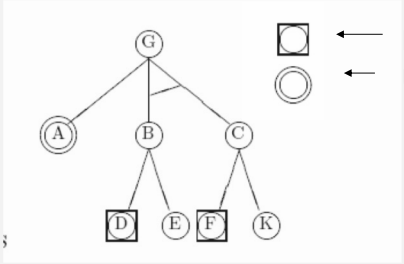
Solution to an PRR problem

- A sub-graph s_q of an AND-OR graph is said to be solution to a problem q if
  - s_q is rooted at q
  - Each non-leaf node y in s_q, has exactly one connector out of it that belongs to s_q
  - Each leaf node in s_q is a primitive problem (i.e. a member of P)
- A problem q is said to be solvable if
  - a sub-graph s_q of an AND-OR graph is a solution to q
- Solving a problem G using a PRR (G, O, P) entails finding a sub-graph S_G of the corresponding AND-OR graph that is a solution of G

Question – How can we solve an PRR problem?

- Basic idea:
  - Generalize state-space search
- How?
  - Generalize partial paths to sub graphs of the PRR AND-OR graph
  - Expanding a node must comply with the semantics of AND and OR connectors
  - Termination test must comply with the definition of a solution

Example – BFS



Primitive problem

Unsolvable problem

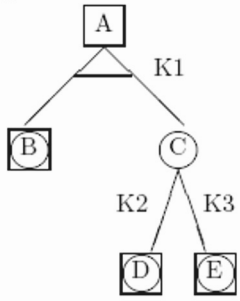
List

(G)
(A) (B & C)
(B & C)
(D & F) (E & F) (D & K) (E & K)

Exercise: Solve the same problem using DFS

Optimal (minimum cost) solution of AND-OR graphs

Example:



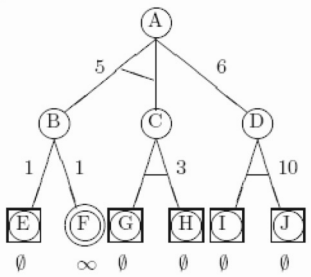
Cost of an unsolvable primitive problem – infinity

Cost of connectors and primitive problems are assumed to be strictly positive and bounded

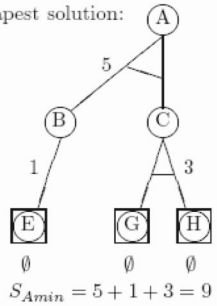
$$Cost(S_A) = Cost(k_1) + Cost(S_B) + Cost(S_C)$$

Optimal solution of an SRR problem

Example:

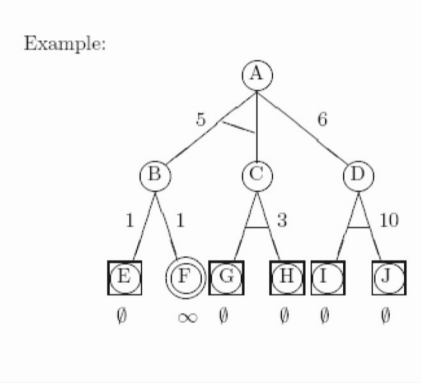


Cheapest solution:



$S_{Amin} = 5 + 1 + 3 = 9$

Branch and Bound Search for Optimal Solution



List

(A)

(B & C) (D)

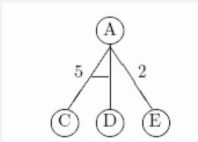
(E & C) (D) (F & C)

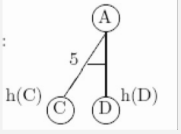
(D) (E & G & H) (F & C)

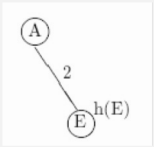
(E & G & H) (I & J) (F & C)

$$Cost(A) = Cost(E \,\&\, G \,\&\, H) = 5 + 1 + 3 + 0 + 0 = 9$$

Using Heuristics







$$f(C \,\&\, D) = Cost(A) + 5 + h(C) + h(D)$$

$$f(E) = Cost(A) + 2 + h(E)$$

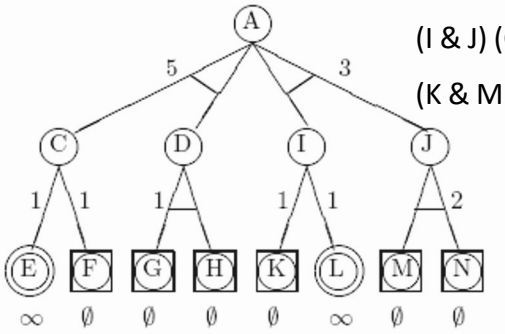
Admissible heuristic function

$h(n) \le h^*(n) = Cost(n)$ ← Cost of the cheapest solution of *n*

AO* - Searching AND-OR graphs

Example:

$h(C) = h(D) = h(I) = h(J) = 1$



(A)

(I & J) (C & D)

(K & M & N) (C & D) (L & M & N)

Properties of AO*

- AO* is a generalization of A* for AND-OR graphs
- AO*, like A*, is admissible if the heuristic function is admissible under the usual assumptions
  - finite branching factor
  - strictly positive action costs
- AO*, like A* is also optimal among the class of heuristic search algorithms that use an additive cost evaluation function