HEURISTICS

Recap: Informed Search

- Define an evaluation function f(n) that estimates how promising a given node is.
- Examples:
  - *UCS: priority queue with f(n) = g(n)*
  - Greedy Best-First: priority queue with f(n) = h(n)
  - A*: priority queue with f(n) = g(n) + h(n)

g(n) = distance from start h(n) = estimate to goal

Today

- Reading
  - Read AIMA Ch. 3.5-3.6 (Informed search)
  - Read AIMA Ch. 4.1-4.2 (Local search)
- Objectives
  - A* and heuristic functions
  - (Introduce local search algorithms)

A^* search: conditions for optimality

- A heuristic h(n) is admissable if it never *overestimates* the cost to the goal

  $0 \leq h(n) \leq h^*(n)$ $h^*(n)$ is true cost to goal

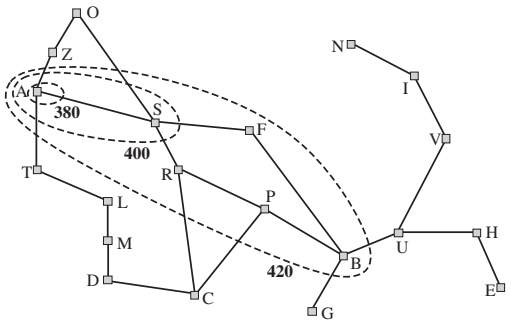
- A heuristic h(n) is consistent if

  $h(n) \leq c(n, a, n') + h(n')$ n' is a successor

- Tree-search version of A^* is optimal if h(n) is admissable
- Graph-search version of A^* is optimal if h(n) is consistent

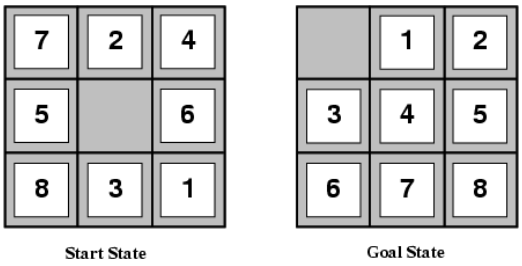
Properties of A* search

- A* expands
  - all nodes with f(n) < C*
  - some nodes with f(n) = C*
  - no nodes with f(n) > C*
- Optimally efficient
- Complete if finite number of nodes with f(n) ≤ C*



Creating admissable heuristic functions

- How do we construct a heuristic function that doesn't overestimate the cost to the goal?



- What are some ideas for heuristic functions?

Creating admissable heuristic functions

- Two-well used heuristics:
  - h_1 = number of misplaced tiles
  - h_2 = sum of the distances of the tiles from goal positions (Manhattan distance)

Goal

	Tiles out of place	Sum of distances out of place
	5	6
	3	4
	5	6

Why are these admissable?

Creating admissable heuristic functions

- Often admissable heuristics are solutions to relaxed problems with fewer restrictions
- A tile can move from square A to square B if
  - A is horizontally or vertically adjacent to B
  - B is blank

Creating admissable heuristic functions

- Often admissable heuristics are solutions to relaxed problems with fewer restrictions
- A tile can move from square A to square B if
  - ~~A is horizontally or vertically adjacent to B~~
  - ~~B is blank~~

relax the rules

- Induces h_1 heuristic, i.e. number of tiles out of place
  - Allows you to pick up the tiles and place in the correct spot

	Number nodes expanded for solution depth d		
	d = 4	**d = 8**	**d= 12**
IDS	112	6384	3.6 million
$A^*(h_1)$	13	39	227

Creating admissable heuristic functions

- Often admissable heuristics are solutions to relaxed problems with fewer restrictions
- A tile can move from square A to square B if
  - A is horizontally or vertically adjacent to B
  - ~~B is blank~~

relax the rules

- Induces h_2 heuristic, i.e. sum of distances to goal position
  - Allows you to move a tile to an adjacent square

	Number nodes expanded for solution depth d		
	d = 4	**d = 8**	**d= 12**
IDS	112	6384	3.6 million
$A^*(h_1)$	13	39	227
$A^*(h_2)$	12	25	73

Creating admissable heuristic functions

expands fewer nodes

	Number nodes expanded for solution depth d		
	d = 4	d = 8	d= 12
IDS	112	6384	3.6 million
$A^*(h_1)$	13	39	227
$A^*(h_2)$	12	25	73

Creating admissable heuristic functions

- Some heuristics are better than others
  - If $h_1(n) \leq h_2(n) \leq h^*(n)$ then h2 dominates h1
  - Manhattan distance dominates tiles out of place
  - A-star search using h_2 will never expand more nodes than A-star search using h_1
  - Can combine admissable heuristics using max

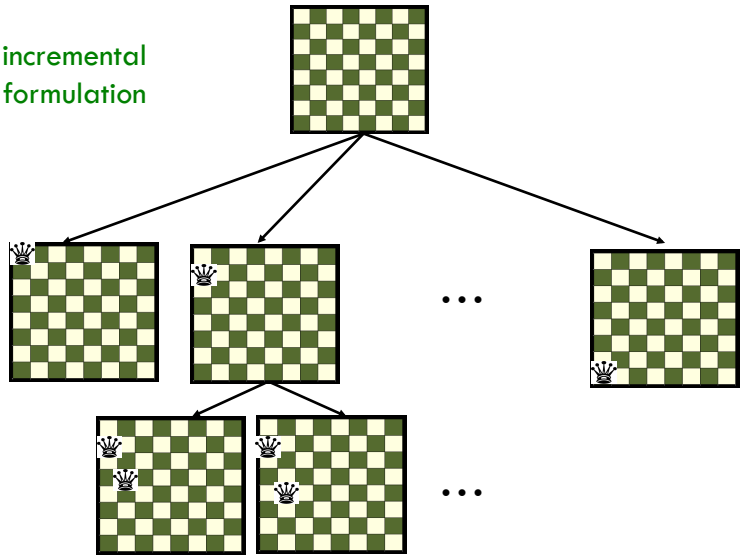
Informed search summary

- Uses additional information to guide search process
  - UCS – cost from start node
  - Greedy best-first – estimate of cost to goal
  - A^* uses both cost from start + estimate to goal
  - A^* is optimal with admissable/consistent heuristic
- A good heuristic is the key!
  - Consider solutions to relaxed problems

Local Search Algorithms

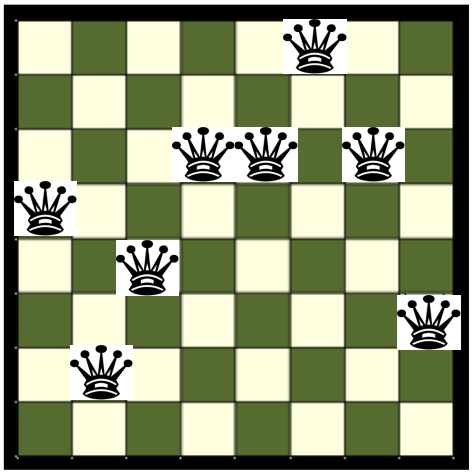
Recall the N-Queens problem

incremental formulation

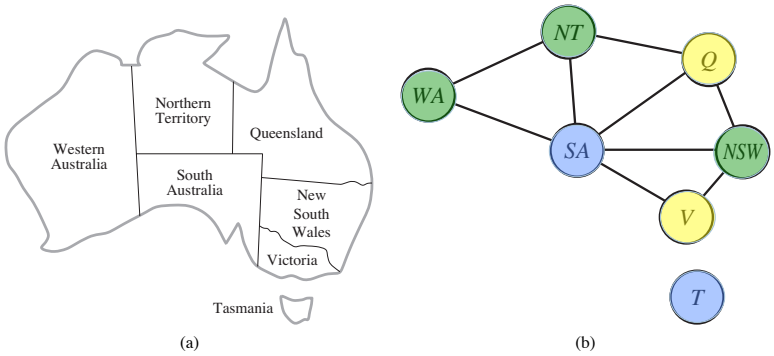


N-Queens alternative approach

complete state formulation



Map Colorings



Local Search Algorithms

- The basic idea:
  1. Randomly initialize state
  2. If not goal state,
     a. make local modification to generate neighbor state OR
     b. enumerate all neighbor states and choose the best
  3. Repeat step 2 until goal state is found (or out of time)
- Requires the ability to *quickly*:
  - Generate a random (probably-not-optimal) state
  - Evaluate the quality of a state
  - Move to other states (well-defined neighborhood function)