Search

Professor Marie Roch

Chapter 3, Russell & Norvig

Solving problems through search

- State – atomic representation of world
- Goal formulation
  - What objective(s) are we trying to meet?
  - Can be represented as a set of states that meet objectives: goal states
- Problem formulation
  - Decide actions and states to reach a goal

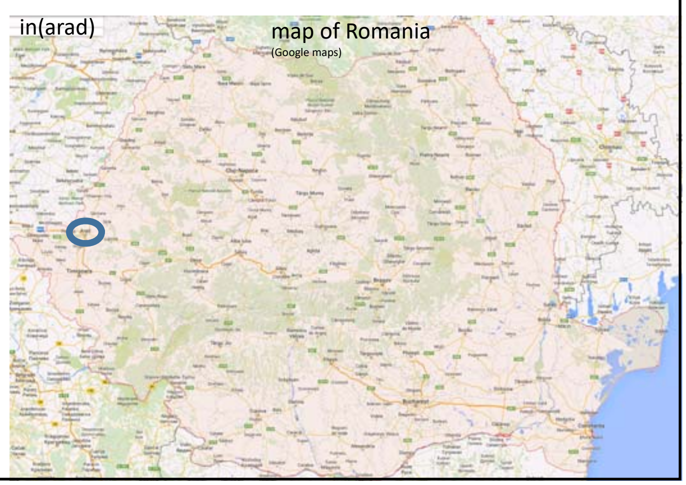
2

Search

- Assume environment is
  - observable
  - discrete (finite # of actions)
  - deterministic actions
- Search process returns a plan:
  set of states & actions to reach a goal state
- Plan can be executed

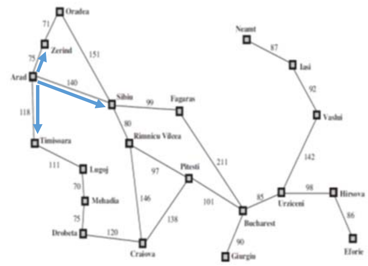
Search problem components

- Initial state



Search problem components

- Initial state
- Actions
  - function that returns set of possible decisions from a given state
  - actions(in(arad)) →{go(sibiu), go(Timisoara), go(zerind)}



Abstract view of Romanian roads
(Russel and Norvig 2010, Fig 3.2)

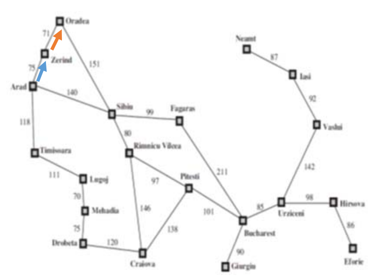
Note: Abstractions are valid when we can map them onto a more detailed world

5

Search problem components

- Initial state
- Cost
  - Each action has a step cost:
    cost(in(arad), go(zerind), in(zerind)) = 75
  - A path has a cost which is the sum of its step costs:

    path: in(arad), in(zerind), in(Oradea)
    has cost: 146 (75+71)

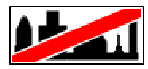


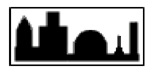
Abstract view of Romanian roads
(Russel and Norvig 2010, Fig 3.2)

6

Search problem components

- Initial state
- Actions
- Cost
- Transition model
  Function that reports the result of an action applied to a state:
  result(in(arad),go(zerind) →in(zerind)


Arad


Zerind

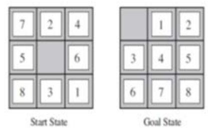
Search problem components

- Initial state
- Actions
- Cost
- Transition model
- Goal predicate
  Is the new state a member of the goal set?
  goal: {in(bucharest)}

Any path that reaches a goal is a *solution*, the lowest cost path is an *optimal solution*.

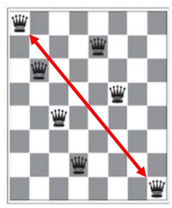
Sample toy problem

- n-puzzle



8-puzzle and one possible goal state
[Figure 3.4 R&N 2010]

- n-queens



8-queens state
[Figure 3.5 R&N 2010]

see text for other examples

9

Constructing a problem: n-queens

- States
  1. complete-state:
     - n-queens on board
     - move until no queen can capture another.
  2. Incrementally place queens
     - initial empty board
     - add one queen at a time

10

Incremental n-queens

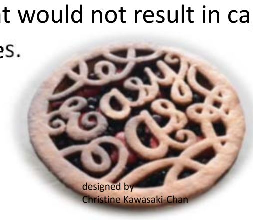
- state: Any arrangement of [0,n] queens
- initial state: empty board
- actions: add queen to empty square
- transition model: new state with additional queen
- goal test: n queens on board, none can attack one another

Incremental n-queens

- A well-designed problem restricts the state space
  - Naïve 8 queens
    1st queen has 64 possibilities
    2nd queen has 63 possibilities…

    $$64 \times 63 \times 62 \ldots \approx 1.8 \times 10^{14}$$

  - Smarter:
    - Actions only returns positions that would not result in capture
    - State space reduced to 2057 states.



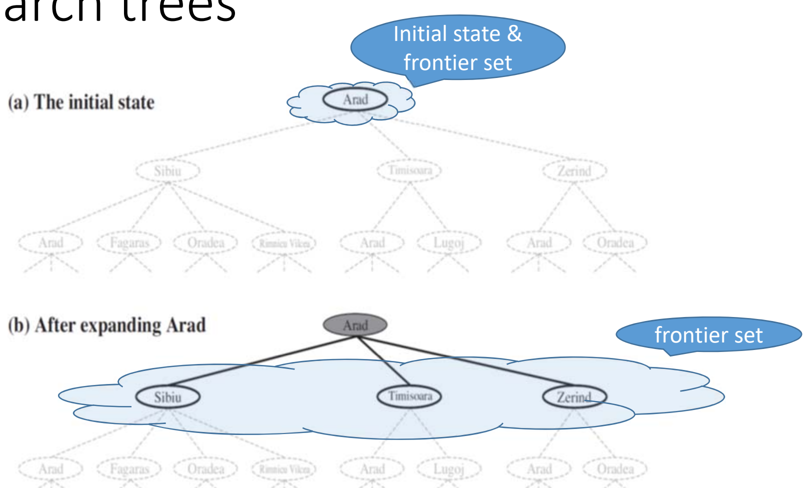
Classic real-world problems

- route-finding problem
  - transportation (car, air, train, boat, …)
  - networks
  - operations planning
- touring problem
  Visit a set of states ≥1 time
- traveling salesperson
  Visit a set of state 1 time

- Others: VLSI layout, autonomous vehicle navigation & planning, assembly sequencing, pharmaceutical discovery

13

Search trees

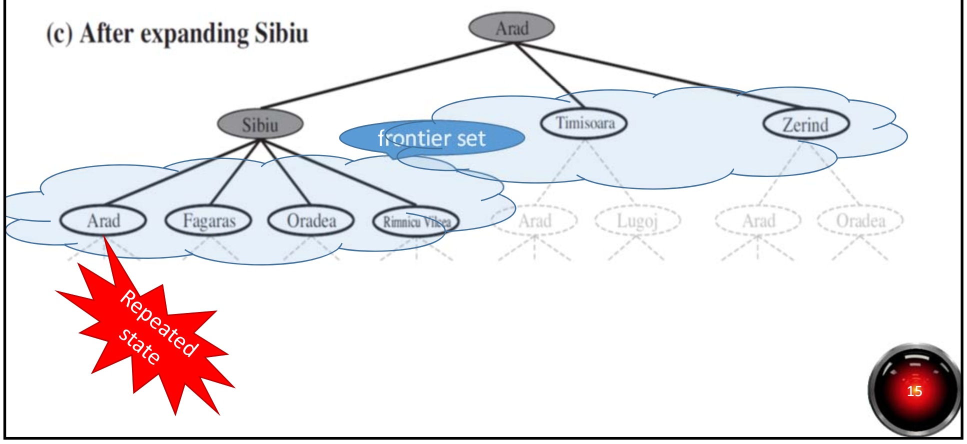


[Figure 3.6 R&N 2010]

frontier set also known as an open list

14

Search trees



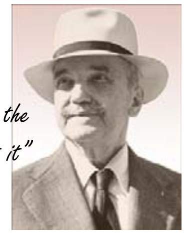
Search tree

- Frontier set consists of leaf nodes
- Redundant paths occur when
  - $\exists$ more than 1 path between a pair of states
  - cycles in the search tree (loops) are a special case

16

Redundant paths

"Those who cannot remember the past are condemned to repeat it"



George Santayana,
Spanish-American philosopher 1863-1952

- Sometimes, we can define are our problem to avoid cycles
  e.g. n-queens: queen must be placed in the leftmost empty column
- Otherwise: Explored set
  - Track states that have been investigated
  - Don't add any actions that have already occurred

Tree Search

```
function tree-search(problem)
  frontier = problem.initial_state()
  done = found = False
  while not done
    node = frontier.get_node()  # remove state
    if node in problem.goals()
      found = done = True
    else
      frontier.add_nodes(results from actions(node))
      done = frontier.is_empty()
  return solution if found else return failure
```

Graph Search

```
function graph-search(problem)
   frontier = problem.initial_state()
   done = found = False
   explored = {} # keep track of nodes we have checked
   while not done
      node = frontier.get_node()  # remove state
      explored = union(explored, node)
      if node in problem.goals()
         found = done = True
      else
         # only add novel results from the current node
         nodes = setdiff(results from actions(node), union(frontier,explored))
         frontier.add_nodes(nodes)
         done = frontier.is_empty()
   return solution if found else return failure
```

19

Search architecture

- Node representation
  - state
  - parent – ancestor in tree
    allows us to find the solution from a goal node by chasing pointers and reversing the path
  - action – Which action was used on parent to generate this node
  - path-cost – What is the cost to reach this node from the tree's root. Usually denoted g(n).

20

Search architecture

```
function child-node(problem, node, action)
  child.state = problem.result(node.state, action)
  child.parent = node
  child.path_cost = node.path_cost +
    problem.cost(node.state, action, child.state)
  return child
```

21

Search architecture

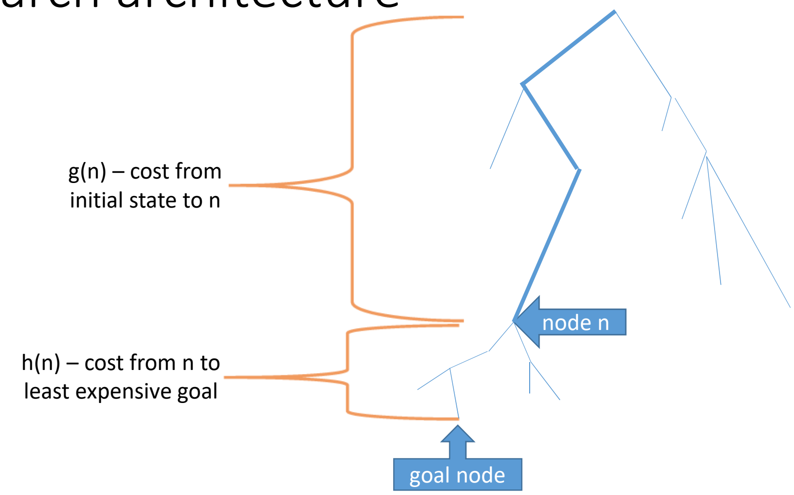
- frontier set is usually implemented as a queue
  - FIFO – traditional queue
  - LIFO – stack
  - priority

  We will develop a way such that it can always be a priority queue.

- Explored set – Need to make states easily comparable
  - hash the state or
  - store in canonical form (e.g. sort visited cities for traveling salesperson problem)

22

Search architecture



g(n) *and h(n) are frequenty not known precisely.*
Estimates are denoted or $g'(n)$ *&* $h'(n)$ *or* $\hat{g}(n)$ *&* $\hat{h}(n)$

A generic graph search algorithm

```
function tree-search(problem)
    frontier = problem.initial_state()  # priority queue based on lowest cost
    done = found = False
    explored = {} # keep track of nodes we have checked
    while not done
        node = frontier.get_node()  # remove state
        explored = union(explored, node)
        if node in problem.goals()
            found = done = True
        else
            # only add novel results from the current node
            nodes = setdiff(results from actions(node), union(frontier,explored))
            for n in nodes
                estimate a cost g'(n) + h'(n)
            frontier.add_nodes(nodes)  # merge new nodes in by estimated cost
            done = frontier.is_empty()
    return solution if found else return failure
```

Uninformed (blind) search

- No awareness of whether or not a state is promising
- Strategies depend on order of node expansion
  - breadth-first
  - uniform-cost
  - depth-first
  - variants: depth-limited, iterative deepening, bidirectional
- Note: Text uses different queue types for frontier, with generic search algorithm everything is a priority queue, smallest values first.

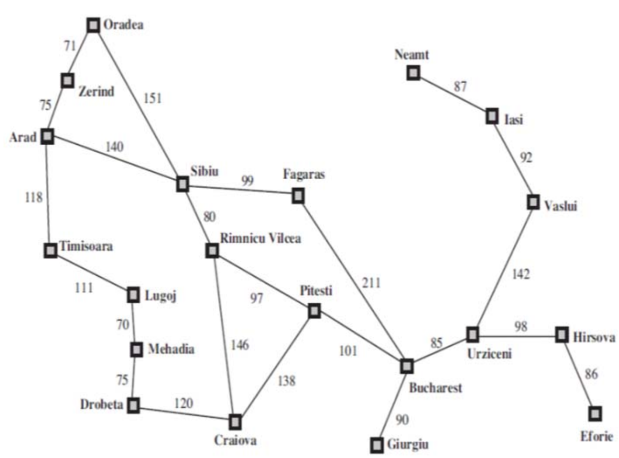
25

Breadth-first search

- $\forall n\ g'(n) = \text{depth}(n)$ and $h'(n) = k$ (e.g. $k=0$)

Abstract view of Romanian roads

(Russel and Norvig 2010, Fig 3.2)



26

Breadth-first search

- Guarantees
  - completeness – will find a solution
  - best path *if cost is a nondecreasing f(depth)*
- How can we measure performance?
  - Time complexity
  - Space complexity

27

Complexity (CS 310 material)

- Measure of the number of operations (time) or memory (space) required
- Analysis of performance as the number of items n grows:
  - worst case
  - average case
- Example:

There are $T(n)=4n^2+1$ arithmetic operations

```
def foobar(n):
  x = 0
  for i in xrange(n):
    for j in xrange(n):
      x = x + i*i + j*j
  return x * x
```

28

Complexity

- We defined "big oh" of n as follows:

$$T(n) \text{ is } O\big(f(n)\big) \text{ if } T(n) \leq kf(n)$$

$$\text{for some } k \ \& \ \forall n > n_0$$

- Role of k and n_0

  Coefficients of highest order polynomial aren't relevant.

- Implications:
  - T(n) = $4n^2+1$ → $O(n^2)$
  - T'(n) = $500n+8$ → O(n)

  For some small values of n, T(n) is better, but as n increases T(n) will be worse. Using the big-oh notation abstracts this away and we know in general that the second algorithm is better.

29

Search complexity

Measured with respect to search tree:

- Complexity is a function of
  - Branch factor – max # of successors
  - Depth of the shallowest goal node
  - Maximum length of a state-space path
- Time measurement: # nodes expanded
- Space measurement: maximum # nodes in memory

30

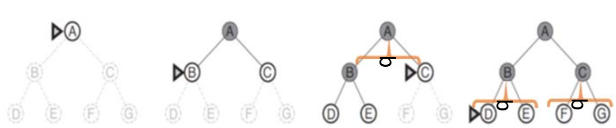
Search complexity

- "Search cost" – time complexity
- "Total cost" – time and space complexity
  Problematic to fuse the metrics…

31

Breadth-first search performance

- Assume branch factor b
- Time complexity:
  $b + b^2 + b^3 + \cdots + b^d = O(b^d)$ *
- Space complexity
  - Every generated node remains in memory, $O(b^{d-1})$ in explored and $O(b^d)$ in frontier.



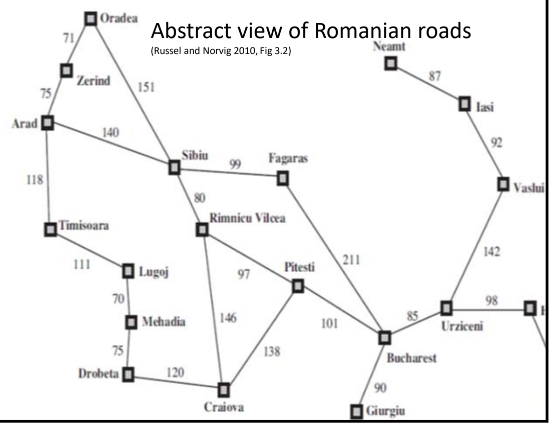
32

* See text for discussion of $O(b^d)$ vs. $O(b^{d+1})$

Uniform-cost search

- Similar to breadth-first, g’(n) uses edge costs
- $\forall n\ g'(n) = g(n)$ and $h'(n) = k$
- Nodes are expanded in order of optimal cost → optimal solution
- Complexity function of minimum cost for all actions

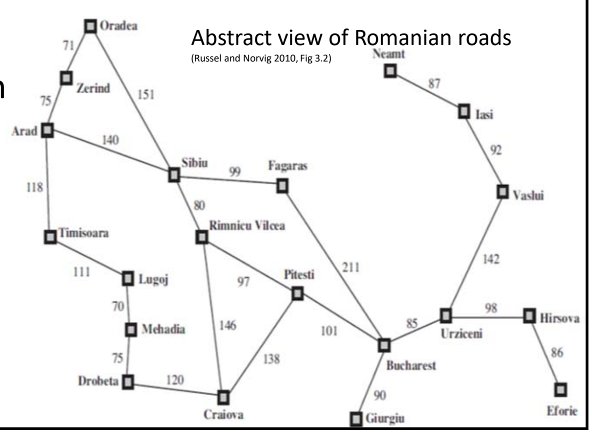
Abstract view of Romanian roads
(Russel and Norvig 2010, Fig 3.2)



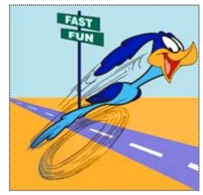
Depth-first search

- Deepest node is expanded first
- $\forall n\ g'(n) = k$ and $h'(n) = -depth(k)$
- Non-optimal
- Incomplete search
- Why bother?

Abstract view of Romanian roads
(Russel and Norvig 2010, Fig 3.2)



Depth-first search (DFS)



- DFS will explore other paths when there are no successors.
- Fast! If you hit the right path… but the average case analysis is $O(b^m)$ where m is maximum depth.
- Space complexity is better: $O(bm)$

35

Iterative deepening

- Prevents infinite loops of depth-first search
- Basic idea
  - Depth-first search with a maximum depth
  - If the search fails, repeat with a deeper depth

36

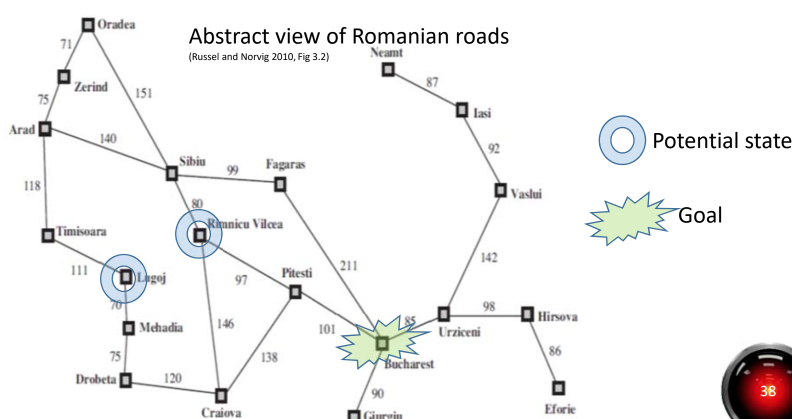
Uninformed search

- Other variants exist
- For large search spaces, it is generally a bad idea

37

Informed, or heuristic, search

- General idea: Can we guess the cost to a goal based on the current state?



38

Heuristic

- h(n) – Actual cost from a search graph node to a goal state *along the cheapest path*.

- h'(n) – An estimate of h(n), known as a heuristic.

Note that your text does not make a notational distinction between the actual cost and the estimated one and always uses h(n), so we will frequently follow suit.

39

Heuristic

- h(n) is always ≥ 0
- h(n) is problem specific
- Estimators of h(n) are similar.

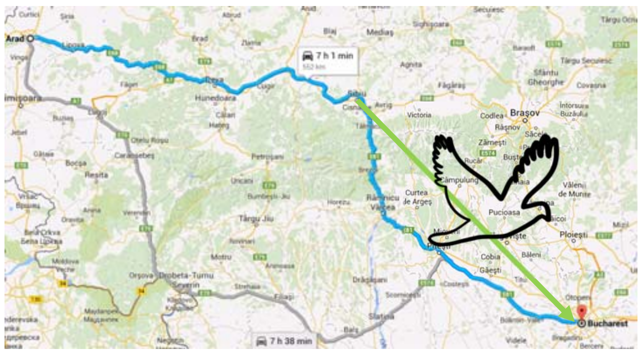
- One can think of a heuristic as an educated guess. We will look at how to construct these later…

40

Greedy best-first search

- g(n) = 0, h(n) is heuristic value
- Example h(n) for Romania example:

  as the bird flies distance



41

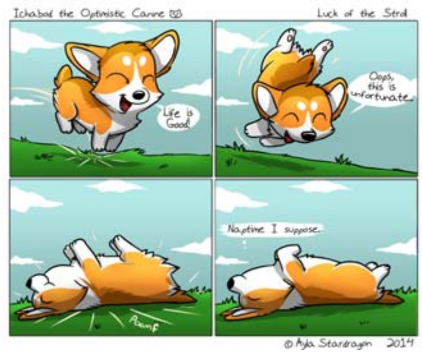
A* Search

- “A-star” search uses:
  - g(n) = cost incurred to n
  - h(n) = estimate to goal

  A* is the estimated cost form start to goal through state n

42

Heuristic properties

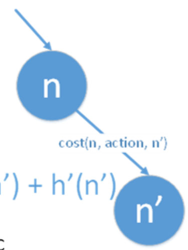


- admissible – h'(n) is admissible if it *never overestimates* the cost to goal

  One can think of it as optimistic: h'(n)

- consistency (aka monotonicity) – h'(n) is consistent if

consistency:
$h'(n) \leq cost(n, action, n') + h'(n')$



Note: We are being careful about distinguishing the heuristic estimator h'(n) from the actual distance h(n)

43

Heuristic properties

- Every consistent heuristic is also admissible.
- A* is guaranteed to be:
  - for trees
    A* optimal if h'(n) is admissible
  - for graphs
    A* optimal if h'(n) is consistent

44

Understanding A*

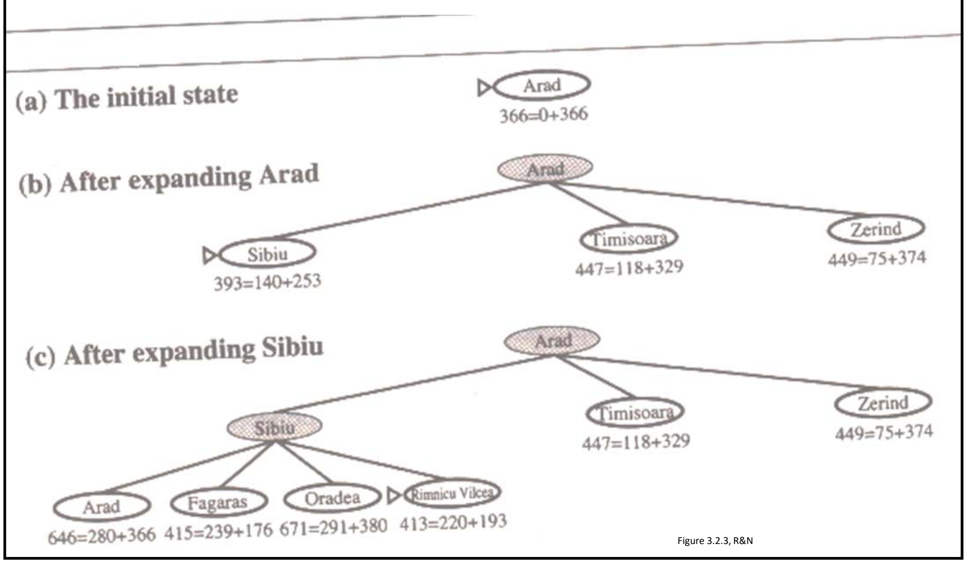
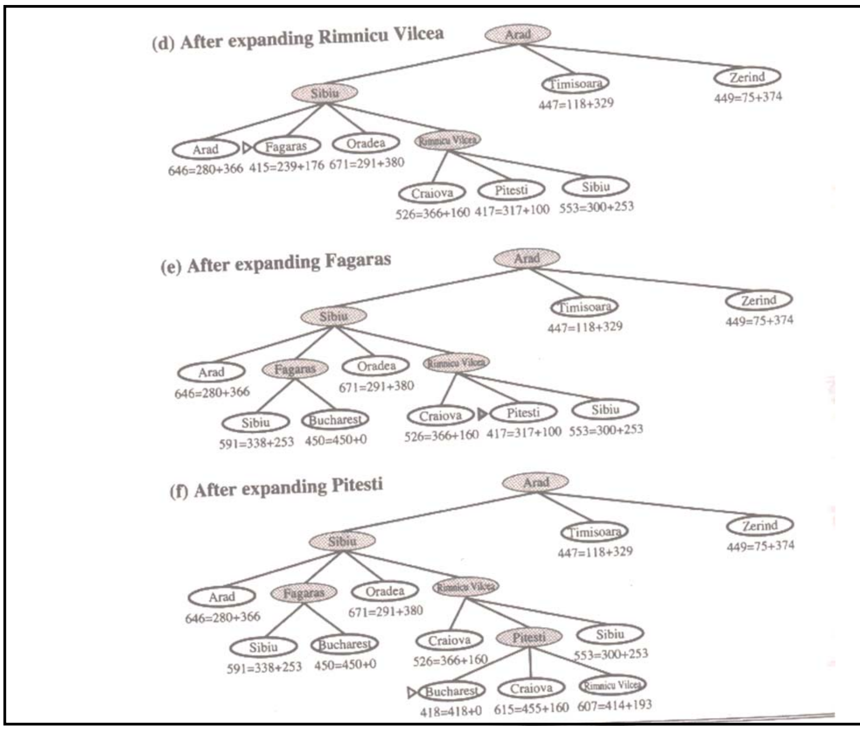


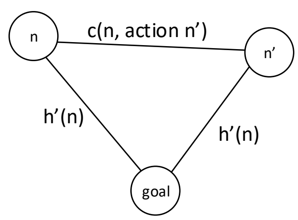
Figure 3.2.3, R&N



46

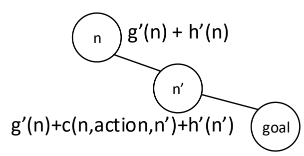
Understanding A* optimality

Consistency revisited: the ▲ inequality – the sum of any two sides ≥ third side



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

If h’ consistent and costs are nonnegative, values of f(n) along any path are *nondecreasing*.



Understanding A* optimality

- Suppose we pick node n
- Is the path to node n’s state optimal?

  Proof by contradiction

  Suppose a better path to the same state is in node b.

  As b and n have the same state, so h(n) = h(b).

  Relative position in queue will be driven by g(n) and g(b).

  If b is better, g(b) < g(n) and we would have picked b first

Understanding A* optimality

- When $h(n)$ is consistent, the properties of:
  - nondecreasing values of $f(n)$
  - guarantee that we pick the best path to n

  ensure that the first goal node we find is optimal.

- Completeness holds when there are a finite number of nodes with $f(n) <$ the optimal cost

Limitations of A*

- Need to find a heuristic
- Show it is consistent (for graph search) if optimal goal is required.
- Show the graph is finite for nodes with cost lower than the optimal one if completeness is required

- Note: expanded set requires nodes in memory and is a frequent limitation of A*

A* variants

- iterative deepening A*
  Same idea as iterative depth-first search,
  but we limit on f(n)

- SMA* - simplified memory A*
  - When memory is full
    - drops worst frontier node (highest f(n))
    - stores that value in parent, and will only reconsider branch when everything looks worse than the stored value
  - Details beyond our scope

51

Heuristic search summary

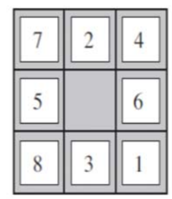
- A* can still have problems with space complexity
  - iterative deepening A*
  - other alternatives listed in text

- Complexity of A* search is tricky, but is related to
  - the error in the heuristic, h(n)-h'(n)
  - and solution depth

52

Developing heuristics

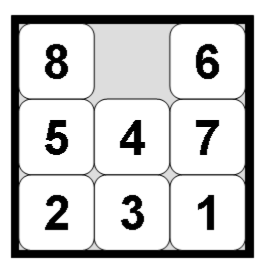
- Requires
  - knowledge of problem domain
  - thinking a bit (usually)
- Effort to show that heuristic is
  - admissible
  - consistent
- What heuristics could we use for the N-puzzle



Start State

N-puzzle heuristics

- Common heuristics
  - $h_1(n)$ – Number of misplaced tiles
  - $h_2(n)$ – Sum of Manhattan[1] distance of tiles to solution
- Are these
  - admissible? (never overestimates)
  - consistent? (non-decreasing path cost)



[1] Also known as city-block distance, the sum of vertical and horizontal displacement.

Heuristics and performance

- Branching factor
  - Measured against a complete tree of solution depth d
  - Suppose A* finds a solution at
    - depth 5
    - 52 nodes expanded (53 with root)
  - A complete tree of depth 5 would have
    $$52 + 1 = b^* + (b^*)^2 + (b^*)^3 + (b^*)^4 + (b^*)^5$$
    where b* is the branch factor
  - Using a root finder for $1(b^*)^5 + 1(b^*)^4 + 1(b^*)^3 + 1(b^*)^2 + 1(b^*)^1 - 53(b^*)^0 = 0$ we see b*≈1.92

55

Heuristics and performance

- 8-puzzle example averaged over 100 instances

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

- branch factors closer to one are better

56

Finding heuristics

- Okay, developing a heuristic is hard
- Can we make it easier?

57

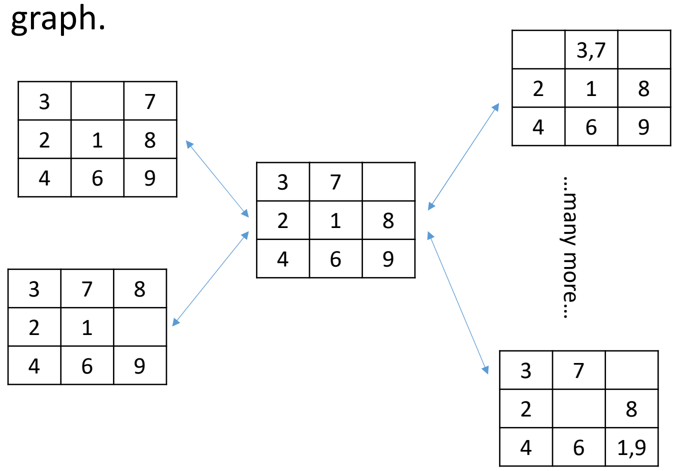
Relaxed problem heuristics

- Let's return to the N-puzzle
- Suppose we allowed
  - A tile to move onto the next square regardless of whether or not it was empty.
  - A tile to move anywhere.
- These are relaxations of the rules

58

Relaxed problems

We can think of these as expanding the state space graph.



Relaxed problem heuristics

- The original state space is a subgraph of the new one.
- Heuristics on relaxed state space
  - Frequently easier to develop
  - If admissible/consistent properties hold in relaxed space, they also hold in the problem state space.

Relaxation

- Can specify problem in a formal language, e.g.
  - move(A,B) if
    - verticalAdjacent(A,B) and
    - horizontalAdjacent(A,B) and
    - isempty(B)
- Possible relaxations
  - move(A,B) if adjacent(A,B)
  - move(A,B) if isempty(B)
  - move(A,B)

61

Automatically generated heuristics

With a formal specification of the problem there exist algorithms to find heuristics (beyond our scope, e.g. ABSOLVER)

Machine Learning, 12, 117–141 (1993)

Machine Discovery of Effective Admissible Heuristics

ARMAND E. PRIEDITIS PRIEDITIS@cs.UCDAVIS.EDU
Department of Computer Science, University of California, Davis, CA 95616

62

Multiple heuristics

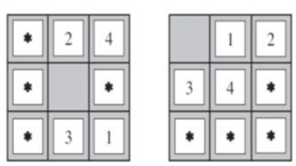
- Regardless of how generated, one may develop multiple heuristics for a problem
- We can merge them

$$h'(n) = \max\left(h'_1(n), h'_2(n), \ldots, h'_i(n)\right)$$

why maximum?

Pattern database heuristics

- Can we solve a subproblem?



Start State　　Goal State

- If we can, we can store its h(n)

Pattern database heuristics

- Cost usually found by searching back from goal nodes.
- Worth it if the search will be executed many times.
- Sometimes patterns are disjoint. If so, the heuristic costs may be added (doesn't work for 8 puzzle)

65

Learning heuristics

- Use experience to learn heuristics
- Beyond our reach for now… (machine learning)

66

Heuristic summary

(rough outline, no substitute for a little thought)