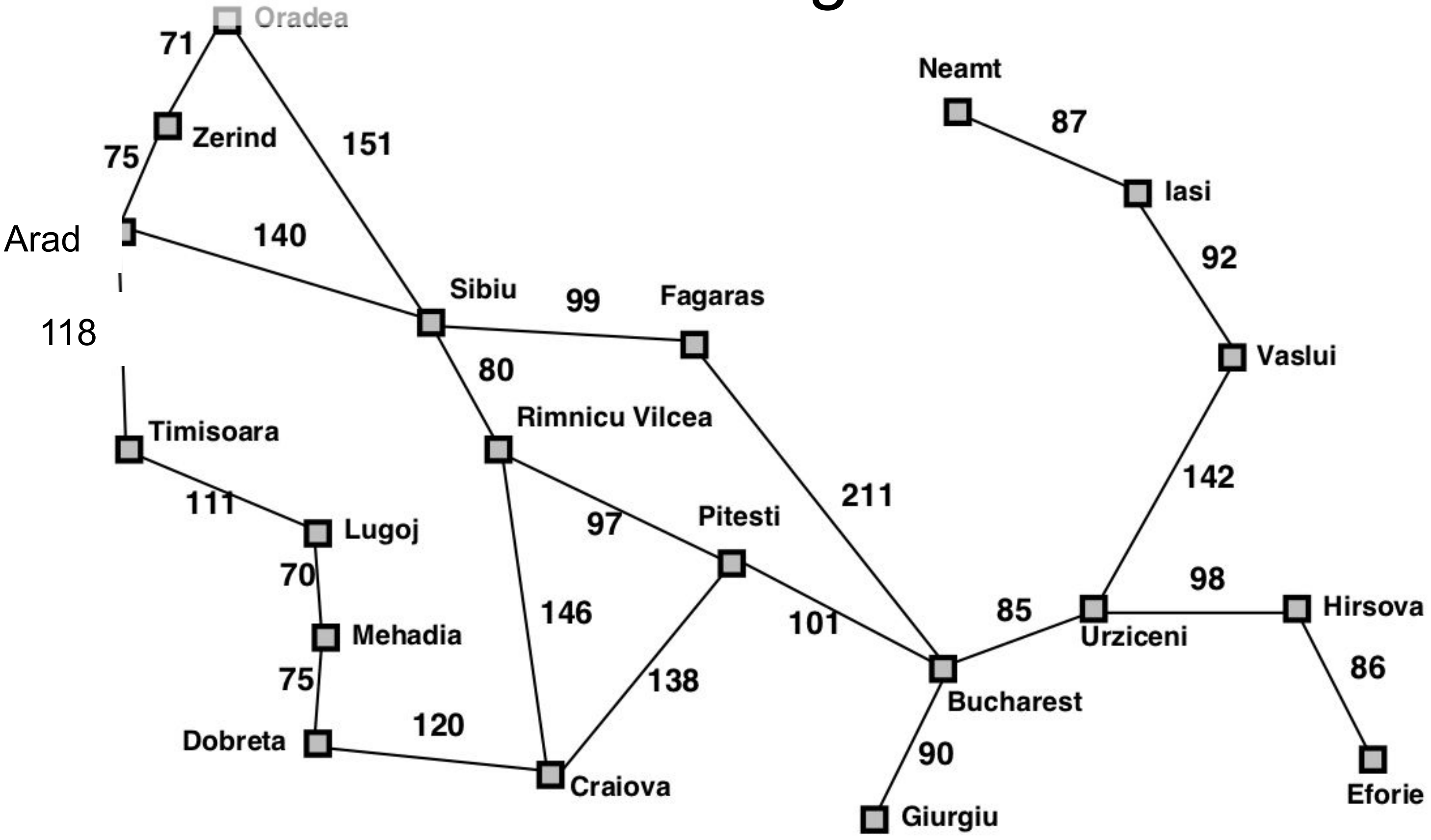
Heuristics

CS 580
Intro to Artificial Intelligence

Romania - A* with straight line distance heuristic



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Romania - A* with straight line distance heuristic

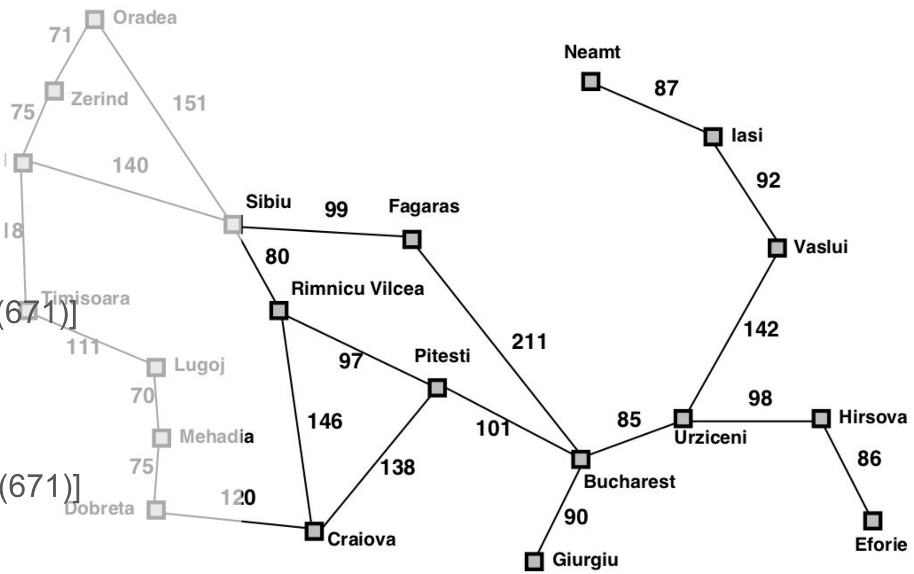
Arad (366)
[**S**(393),**T**(447),**Z**(449)]
[A]

Sibiu (393)
[**R**(413),**F**(415),T(447),Z(449),**O**(671)]
[A,S]

Rimnicu Vilcea (413)
[F(415),**P**(417),T(447),Z(449),**C**(526),O(671)]
[A,S,R]

Fagaras (415)
[P(417),T(447),Z(449),**B**(450),C(526),O(671)]
[A,S,R,F]

Pitesti (417)
[**B**(418),T(447),Z(449),B(450),C(526),**C**(625),O(671)]
[A,S,R,F,P]



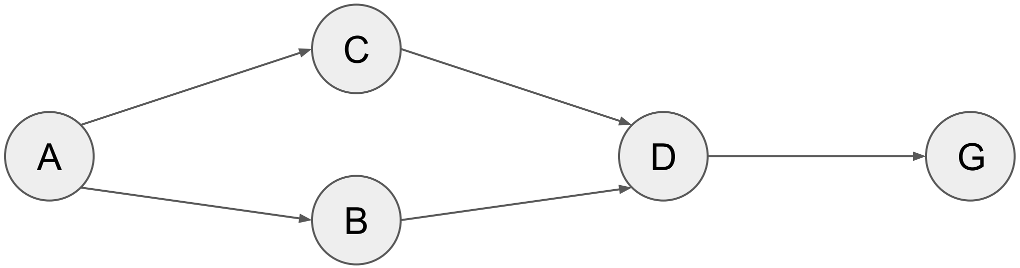
Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Solution: [A->S, S->R, R->P, P->B]
Cost: 418

Why do we need consistency?

- Necessary in the proof to ensure that $f(n_i)$ never decreased
- Why might $f(n_i)$ decrease? Shortcuts!



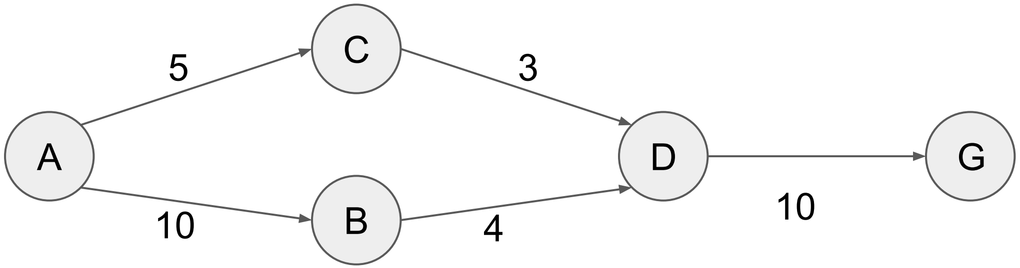
Shortcuts - example (1)

Heuristic: h(A)=15, h(B)=3, h(C)=10, h(D)=0, h(G)=0

Admissible? ✔

Consistent?

h(n) <= c(n,a,n’)+h(n’)
h(A) <= c(A,->,B) + h(B)
15 <= 10+3 ⛔



Shortcuts - example (2)

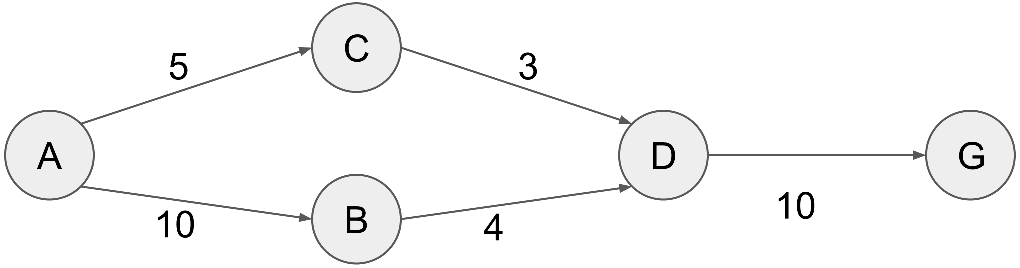
A(15)
[**B(13)**, **C(15)**]
[A]

B(13)
[**D(14)**, C(15)]
[A,B]

D(14)
[C(15), **G(24)**]
[A,B,D]

C(15)
[G(24)]
[A,B,**D**,C]

G(24)
[]
[A,B,D,C,G]



State	A	B	C	D	G
h(State)	15	3	10	0	0

Solution returned: [A->B,B->D,D->G], cost 24
Optimal solution: [A->C,C->D,D->G], cost 18

Fixing Generic Search to handle shortcuts (1)

What was the problem?

- When a shorter path to D was encountered, D was already in the “closed” list
- If we find a shorter path for a node in the closed list, we need to update it’s g(n)...
- And then all the g(n) of the children of of that node…
- Which may re-order the open priority queue…

Yuck

Fixing Generic Search to handle shortcuts (2)

Computationally

- **`update parent of s`** isn't so bad if we use backpointers
- **`recompute g and resort open`** has to trace back the new path!

One implementation

- Do DFS from start state, recompute g as you go
- Don't expand a node if it's already in open

Note for project 1: this is <u>unnecessary</u> as all the heuristics will be consistent or inadmissible anyway

```
Initialize 'current' node to start state
Initialize 'closed' as an empty list
Initialize 'open' as one of (stack, queue, priority queue)
while not( current['state'] is goal state):
     Add current['state'] to closed
     successors = successors of current['state']
     for s in successors:
          if not(s.state is in closed):
               Add new node for state to open
          elif s.cost+current['g']< old cost to s:
               update parent of s
               recompute g and resort open
     current = next node in open that's not in closed
path = list()
while current has a parent:
     Add current['action'] to the front of path
     current = current['parent']
return path
```

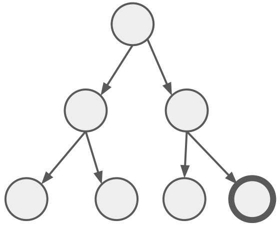
Heuristic effectiveness

The **effective branching factor** (b*) for a heuristic is a way of characterizing how **helpful** that heuristic is.

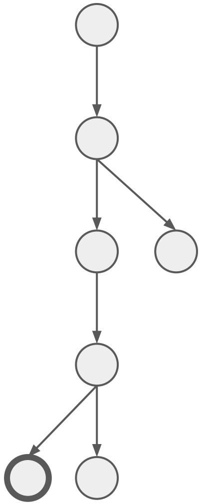
If A* finds a solution at depth *d* expanding *N* nodes, then b* is the branching factor that a uniform tree of depth *d* would need to contain *N*+1 nodes

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

Lower effective branching factor indicates the heuristic will be effective in solving larger problems with reasonable computation time



d=3
N=7
b*=2



d=4
N=7
b*=1.23...

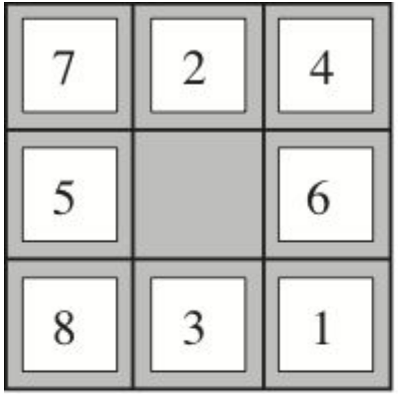
Simple problem domain: 8-puzzle

Initial state: scrambled board
Goal state: tiles in numerical order
Actions: slide one tile into the blank spot (move the blank spot one tile)
Cost: 1 per move

Two heuristics (for comparison):

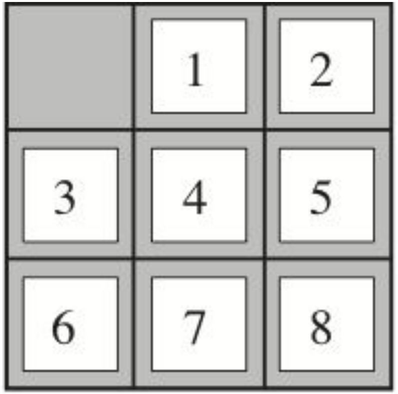
h_1 = # of misplaced tiles

h_2 = distance of all **tiles** to final position



Start State

$h_1 = 8$
$h_2 = 18$



Goal State

$h_1 = h_2 = 0$

Experiment - nodes expanded for 8-puzzle

Depth	2	4	6	8	10	12	14	16	18	20	22	24
IDS	10	112	680	6384	47127	3644035						
A*(h_1)	6	13	20	39	93	227	539	1301	3056	7276	18094	39135
A*(h_2)	6	12	18	25	39	73	113	211	363	676	1219	1641

- 100 random puzzles for each depth
- IDS didn't finish in time for d>12
- Both h_1 and h_2 outperform IDS
- h_2 seems better than h_1 for d>6
- Effective branching factor is relatively stable across problem sizes

Effective branching factor

IDS: 2.45 to 2.87

h_1: 1.33 to 1.79

h_2: 1.22 to 1.79

Is h_2 **always** better than h_1?

Yes!

For any node, $h_1(n) <= h_2(n)$ (each out of place tile must move at least one space)

When comparing heuristics, if $h_a(n) <= h_b(n)$ for all n, we say h_b **dominates** h_a

Since A* with consistent heuristics will always expand every node with $f(n)=g(n)+h(n)<C^*$, we should try to make $h(n)$ as large as possible (still admissible and consistent, efficient to compute)

Admissible heuristics: $0<=h(n)<=h^*(n)$

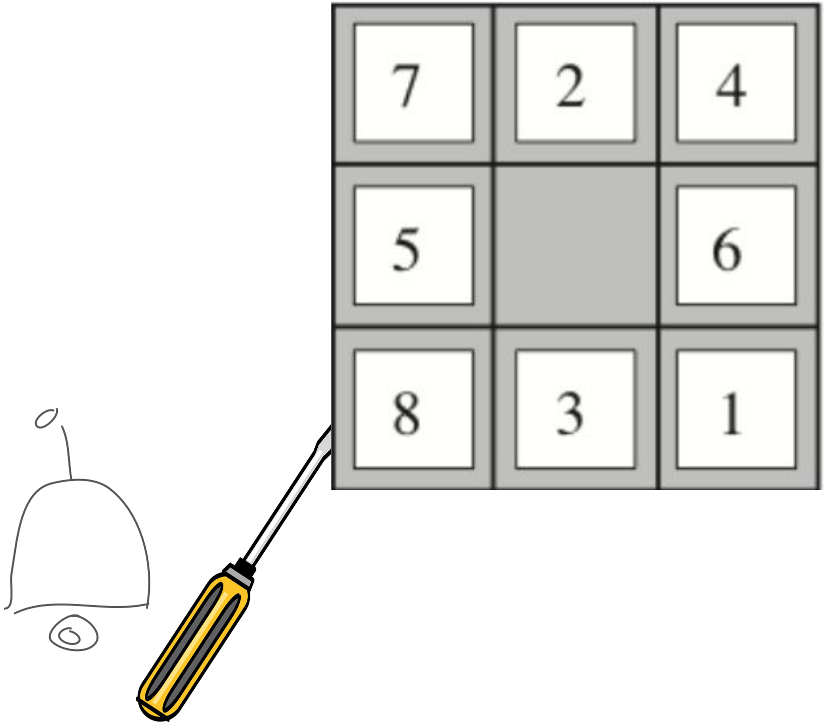
Heuristic design - problem relaxation

One way of generating heuristics is to use solutions to a version of the problem with **fewer constraints**

h_1: path cost if tiles can "teleport" to the correct spot

h_2: path cost if tiles can slide over one another

The relaxed problem has the same state space with additional edges: so the cost of a **solution** in the relaxed problem is **guaranteed** to be an **admissible** heuristic for the original problem



Heuristic design - composite heuristics

If we have a set of (admissible, consistent) heuristics that are non-dominated, we can combine them!

$h(n) = \max\{ h_1(n), h_2(n), \ldots, h_k(n) \}$

Note that this new heuristic dominates* all of the component heuristics

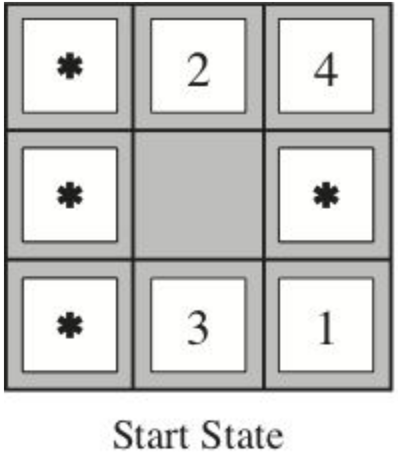
**technically it is non-dominated*

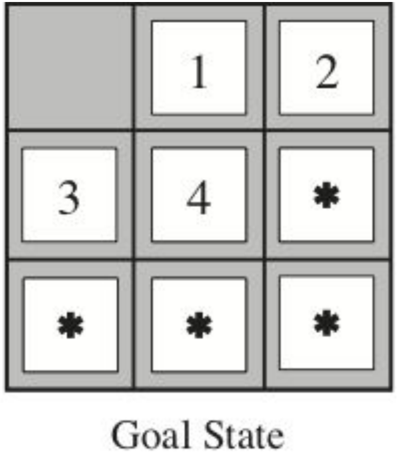
Heuristic design - Pattern databases

Idea: pre-compute the solution to a simpler sub-problem, and **store** the solution length. When searching the larger problem, match states against subproblem patterns and use the solution length as the estimate

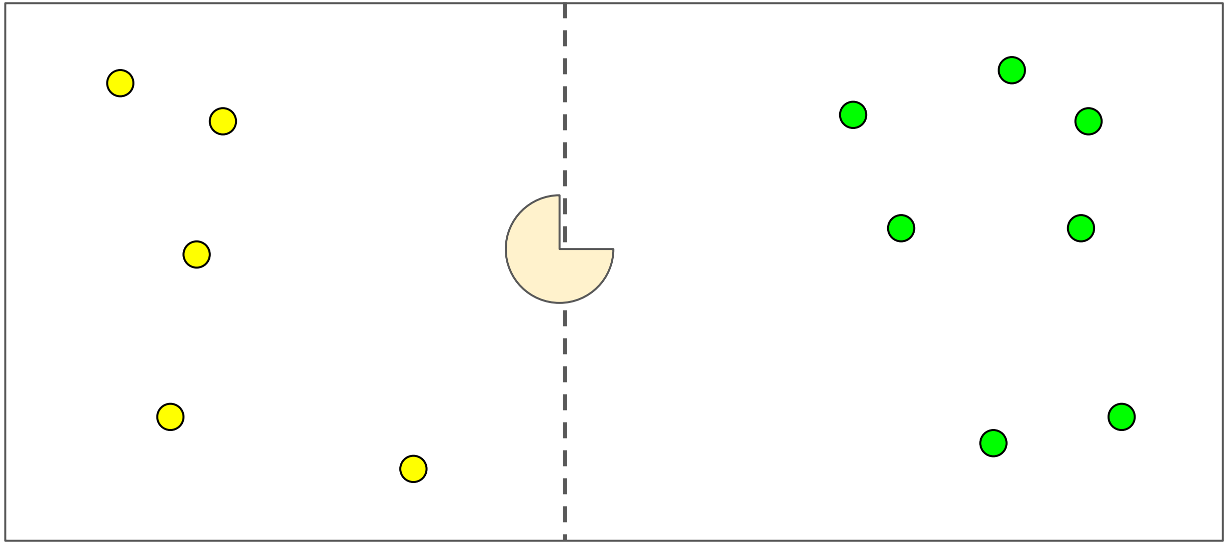
8-puzzle example: solve the puzzle for a subset of the tiles. Different subsets yield different heuristics.

Since sum(h_1,h_2,...) >= max(h_1,h_2,...), why don't we just add heuristics together?





Heuristic design - disjoint pattern databases



If you can split the problem into **disjoint** subproblems, where solving one does not **reduce** the cost of solving another, you can actually **add** the subproblem solution costs (instead of taking the max), but this can be trickier than you expect!

Summary and preview

Wrapping up

- We need consistency to ensure generic search expands in order of increasing f(n). We can fix generic search to work even for inconsistent heuristics, but it can get messy.
- **Effective branching factor** is a useful way of quantifying and comparing heuristic "helpfulness"
- Several ways of designing heuristics: problem **relaxation**, **composite** heuristics, and **pattern databases**.

Next time

- Adversarial Search