CSC384: Intro to Artificial Intelligence

Search

- Chapter 3 of the text is very useful reading. We won't cover the material in section 3.6 in much detail.
- Chapter 4.1, 4.2, some of 4.3 covers heuristic search. We won't talk about the material in sections 4.4, 4.5. But this is interesting additional reading
- Announcements: Prolog Tutorial?

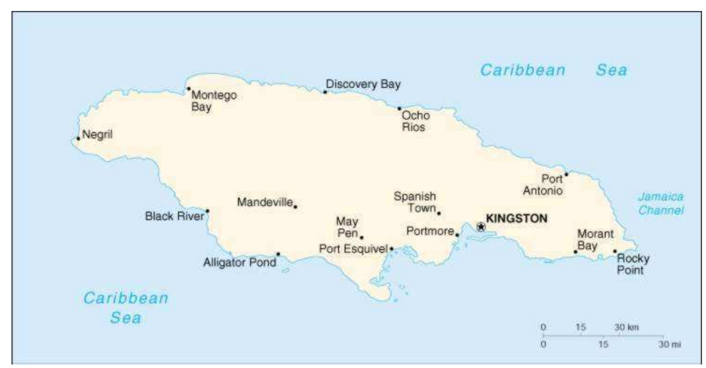
Why Search

- Successful
  - Success in game playing programs based on search.
  - Many other AI problems can be successfully solved by search.
- Practical
  - Many problems don't have a simple algorithmic solution. Casting these problems as search problems is often the easiest way of solving them. Search can also be useful in approximation (e.g., local search in optimization problems).
  - Often specialized algorithms cannot be easily modified to take advantage of extra knowledge. Heuristics provide search provides a natural way of utilizing extra knowledge.
- Some critical aspects of intelligent behaviour, e.g., planning, can be naturally cast as search.

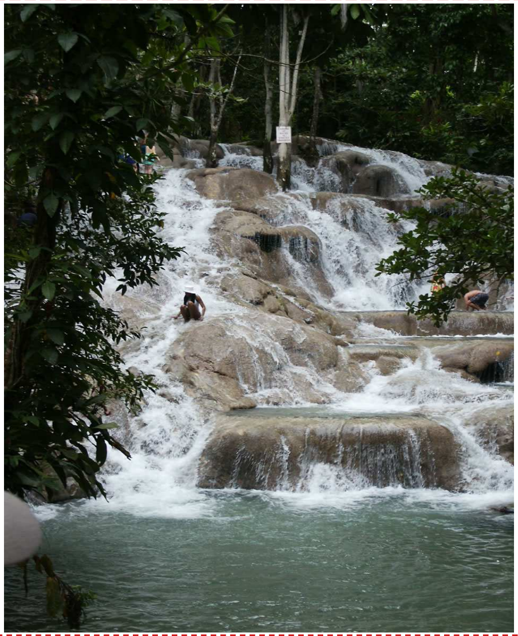
Example, a holiday in Jamaica

Things to consider

- Prefer to avoid hurricane season.
- Rules of the road, larger vehicle has right of way (especially trucks).



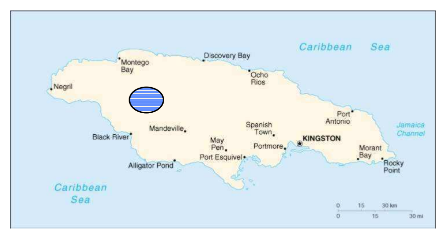
Want to climb up to the top of Dunns river falls.



But you want to start your climb at 8:00 am before the crowds arrive!

Want to swim in the Blue Lagoon

Want to hike the Cockpit Country



No roads, need local guide and supplies.

- Easier goal, climb to the top of Blue Mountain
- Near Kingston. Organized hikes available.
- Need to arrive on the peak at dawn, before the fog sets in.
- Can get some Blue Mountain coffee!

How do we plan our holiday?

- We must take into account various preferences and constraints to develop a schedule.
- An important technique in developing such a schedule is "hypothetical" reasoning.
  - e.g., if I fly into Kingston and drive a car to Port Antonio, I'll have to drive on the roads at night. How desirable is this?
  - If I'm in Port Antonio and leave at 6:30am, I can arrive a Dunns river falls by 8:00am.

How do we plan our holiday?

- This kind of hypothetical reasoning involves asking
  - "what state will I be in after the following sequence of events?"
- From this we can reason about what sequence of events one should try to bring about to achieve a desirable state.
- Search is a computational method for capturing a particular version of this kind of reasoning.

Search

- There are many difficult questions that are not resolved by search. In particular, the whole question of how does an intelligent system formulate its problem as a search problem is not addressed by search.
- Search only shows how to solve the problem once we have it correctly formulated.

The formalism.

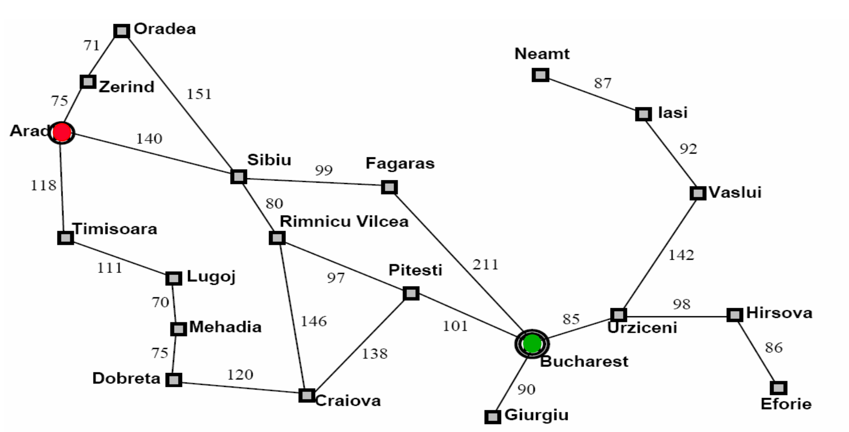
- To formulate a problem as a search problem we need the following components:
  1. Formulate a state space over which to search. The state space necessarily involves abstracting the real problem.
  2. Formulate actions that allow one to move between different states. The actions are abstractions of actions you could actually perform.
  3. Identify the initial state that best represents your current state and the desired condition one wants to achieve.
  4. Formulate various heuristics to help guide the search process.

The formalism.

- Once the problem has been formulated as a state space search, various algorithms can be utilized to solve the problem.
  - A solution to the problem will be a sequence of actions/moves that can transform your current state into state where your desired condition holds.

Example 1: Romania Travel.

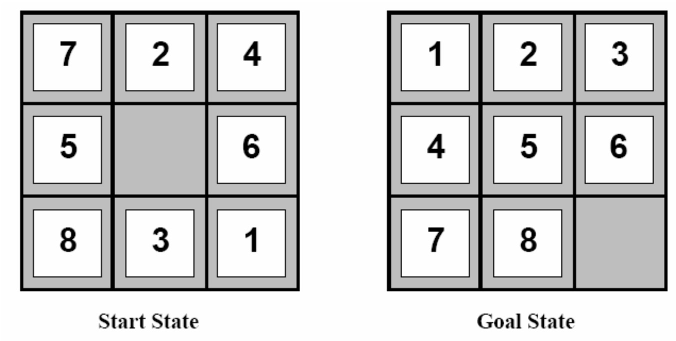
Currently in Arad, need to get to Bucharest by tomorrow to catch a flight. What is the State Space?



Example 1.

- State space.
  - States: the various cities you could be located in.
    - Note we are ignoring the low level details of driving, states where you are on the road between cities, etc.
  - Actions: drive between neighboring cities.
  - Initial state: in Arad
  - Desired condition (Goal): be in a state where you are in Bucharest. (How many states satisfy this condition?)
- Solution will be the route, the sequence of cities to travel through to get to Bucharest.

Example 2. The 8-Puzzle



- Rule: Can slide a tile into the blank spot. (Equivalently, can think if it as moving the blank around).

Example 2. The 8-Puzzle

- State space.
  - States: The different configurations of the tiles. How many different states?
  - Actions: Moving the blank up, down, left, right. Can every action be performed in every state?
  - Initial state: as shown on previous slide.
  - Desired condition (Goal): be in a state where the tiles are all in the positions shown on the previous slide.
- Solution will be a sequence of moves of the blank that transform the initial state to a goal state.

Example 2. The 8-Puzzle

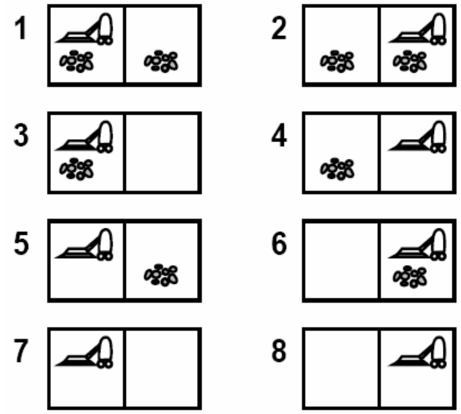
- Although there are 9! different configurations of the tiles (362,880) in fact the state space is divided into two disjoint parts.
- Only when the blank is in the middle are all four actions possible.
- Our goal condition is satisfied by only a single state. But one could easily have a goal condition like
  - The 8 is in the upper left hand corner.
    - How many different states satisfy this goal?

Example 3. Vacuum World.

- In the previous two examples, a state in the search space corresponded to a unique state of the world (modulo details we have abstracted away).
- However, states need not map directly to world configurations. Instead, a state could map to the agent's mental conception of how the world is configured: the agent's knowledge state.

Example 3. Vacuum World.

- We have a vacuum cleaner and two rooms.
- Each room may or may not be dirty.
- The vacuum cleaner can move left or right *(the action has no effect if there is no room to the right/left)*.
- The vacuum cleaner can suck; this cleans the room *(even if the room was already clean)*.

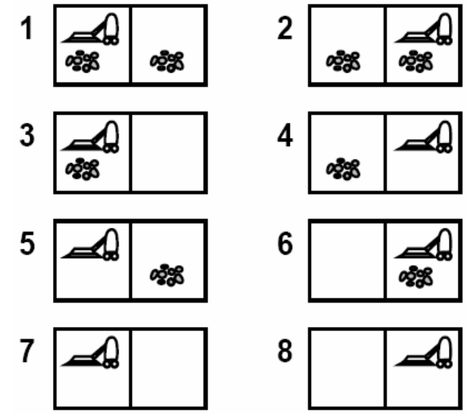


Physical states

Example 3. Vacuum World.

Knowledge level State Space

- The state space can consist of a set of states. The agent knows that it is in one of these states, but doesn't know which.

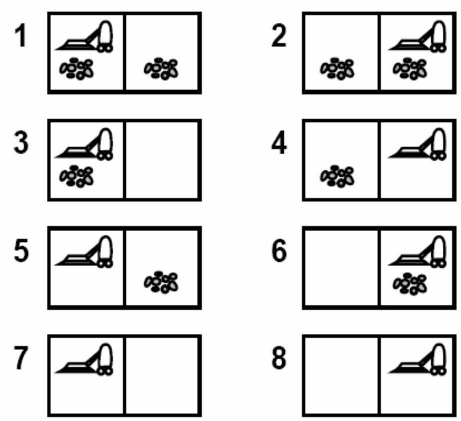


Goal is to have all rooms clean.

Example 3. Vacuum World.

Knowledge level State Space

- Complete knowledge of the world: agent knows exactly which state it is in. State space states consist of single physical states:
- Start in {5}:
  <right, suck>

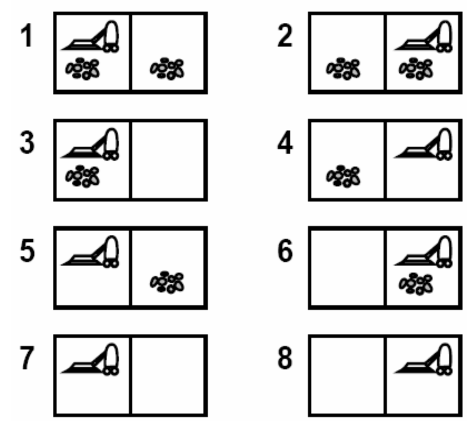


Goal is to have all rooms clean.

Example 3. Vacuum World.

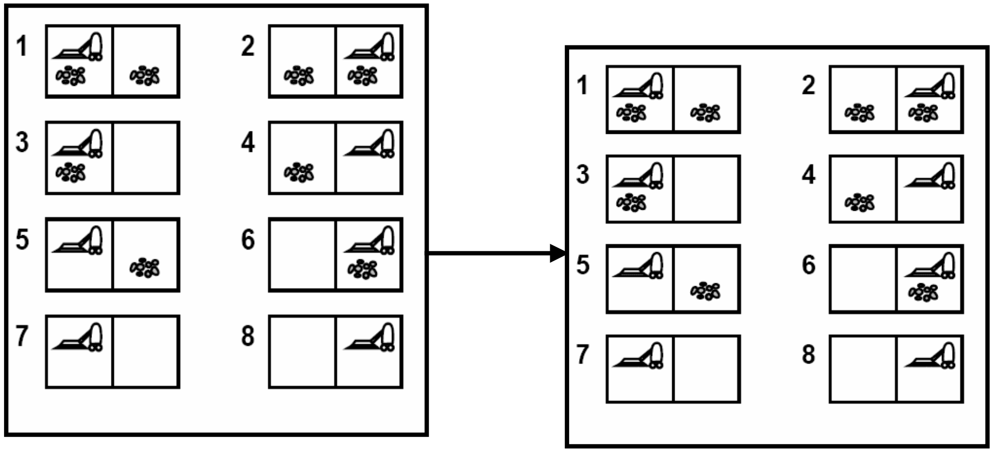
Knowledge level State Space

- No knowledge of the world. States consist of sets of physical states.
- Start in {1,2,3,4,5,6,7,8}, agent doesn't have any knowledge of where it is.
- Nevertheless, the actions <right, suck, left, suck> achieves the goal.



Goal is to have all rooms clean.

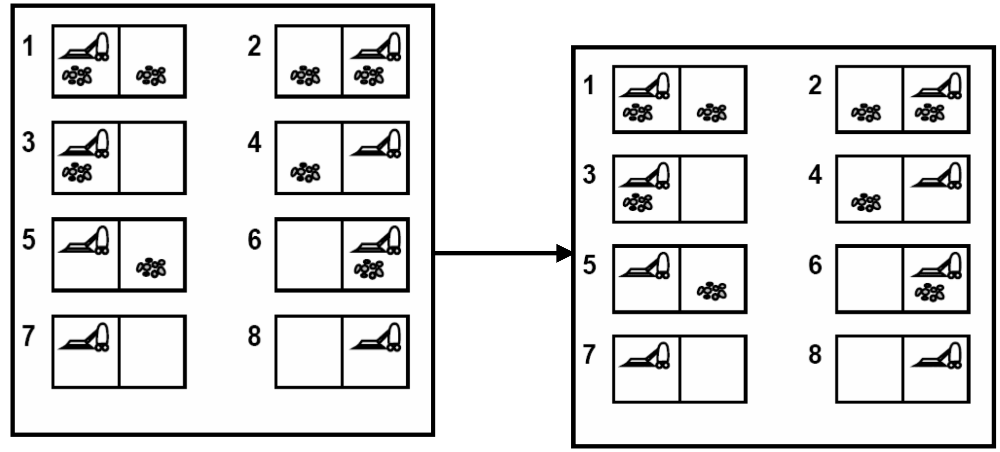
Example 3. Vacuum World.



Initial state.
{1,2,3,4,5,6,7,8}

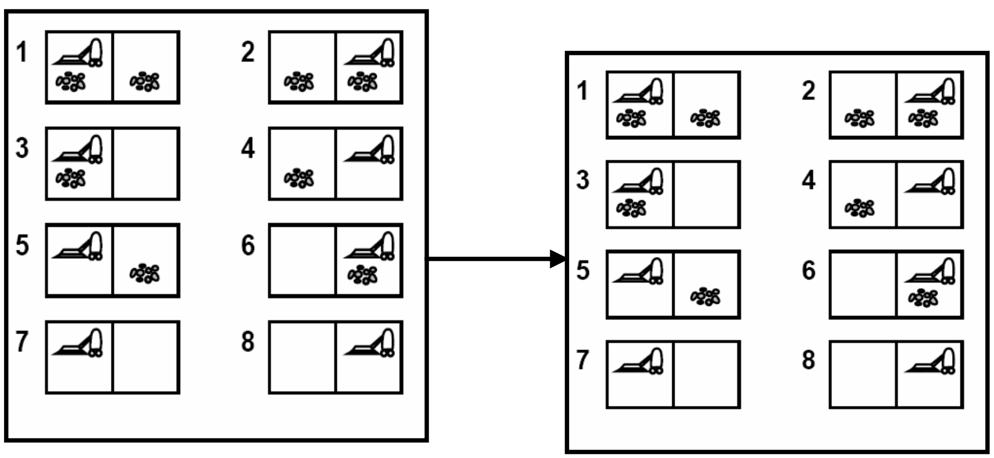
Left

Example 3. Vacuum World.



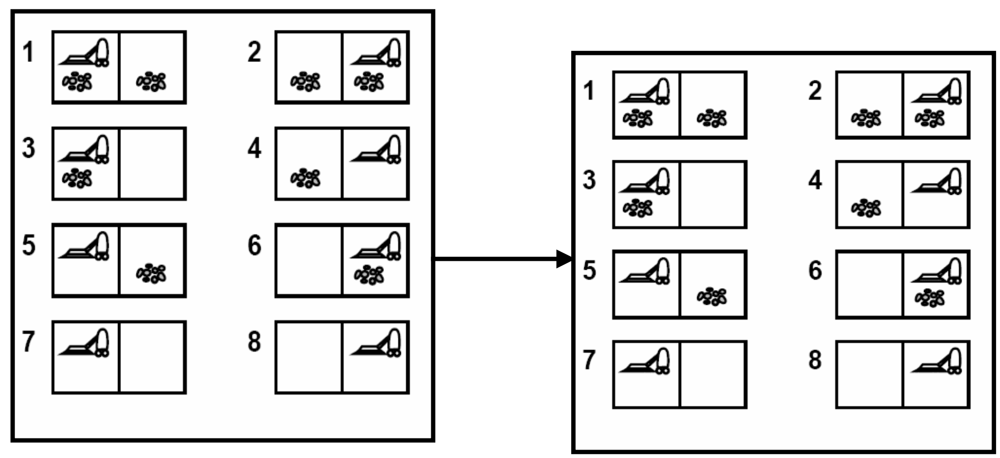
Suck

Example 3. Vacuum World.



Right

Example 3. Vacuum World.



Suck

More complex situations.

- The agent might be able to perform some sensing actions. These actions change the agent's mental state, not the world configuration.
- With sensing can search for a contingent solution: a solution that is contingent on the outcome of the sensing actions
  - <right, **if** dirt **then** suck>
- Now the issue of interleaving execution and search comes into play.

More complex situations.

- Instead of complete lack of knowledge, the agent might think that some states of the world are more likely than others.
- This leads to probabilistic models of the search space and different algorithms for solving the problem.
- Later we will see some techniques for reasoning and making decisions under uncertainty.

Algorithms for Search.

- Inputs:
  - a specified initial state (a specific world state or a set of world states representing the agent's knowledge, etc.)
  - a successor function S(x) = {set of states that can be reached from state x via a single action}.
  - a goal test a function that can be applied to a state and returns true if the state is satisfies the goal condition.
  - A step cost function C(x,a,y) which determines the cost of moving from state x to state y using action a. ($C(x,a,y) = \infty$ if a does not yield y from x)

Algorithms for Search.

- Output:
  - a sequence of states leading from the initial state to a state satisfying the goal test.
  - The sequence might be
    - annotated by the name of the action used.
    - optimal in cost for some algorithms.

Algorithms for Search

- Obtaining the action sequence.
  - The set of successors of a state x might arise from different actions, e.g.,
    - x → a → y
    - x → b → z
  - Successor function S(x) yields a set of states that can be reached from x via a (any) single action.
    - Rather than just return a set of states, we might annotate these states by the action used to obtain them:
      - S(x) = {<y,a>, <z,b>}
        y via action a, z via action b.
      - S(x) = {<y,a>, <y,b>}
        y via action a, also y via alternative action b.

Tree search.

- we use the successor state function to simulate an exploration of the state space.
- Initial call has Frontier = initial state.
  - Frontier is the set of states we haven't yet explored/expanded, and want to explore.

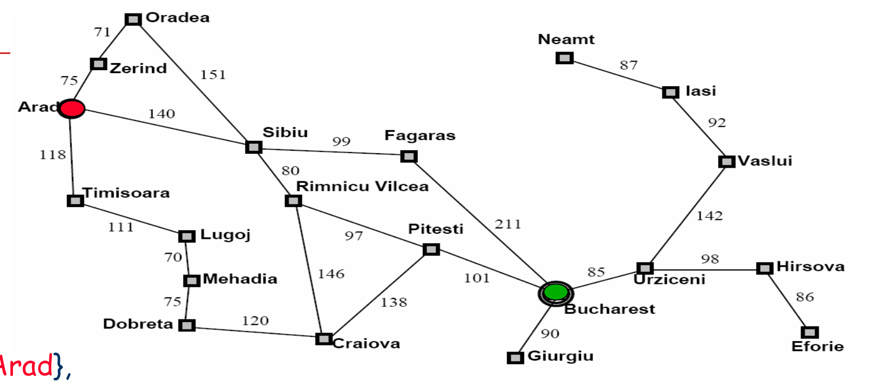
```
TreeSearch(Frontier, Sucessors, Goal? )
  If Frontier is empty return failure
  Curr = select state from Frontier
  If (Goal?(Curr)) return Curr.
  Frontier' = (Frontier – {Curr}) U Successors(Curr)
  return TreeSearch(Frontier', Successors, Goal?)
```


Tree search.

Prolog Implementation:

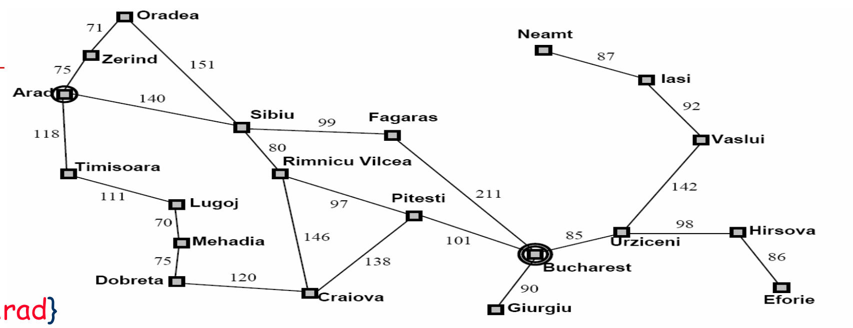
```
treeS([[State|Path],_],Soln) :-
  goal?(State), reverse([State|Path], Soln).

treeS([[State|Path],Frontier],Soln) :-
  genSuccessors(State,Path,NewPaths),
  merge(NewPaths,Frontier,NewFrontier),
  treeS(NewFrontier,Succ,Soln).
```

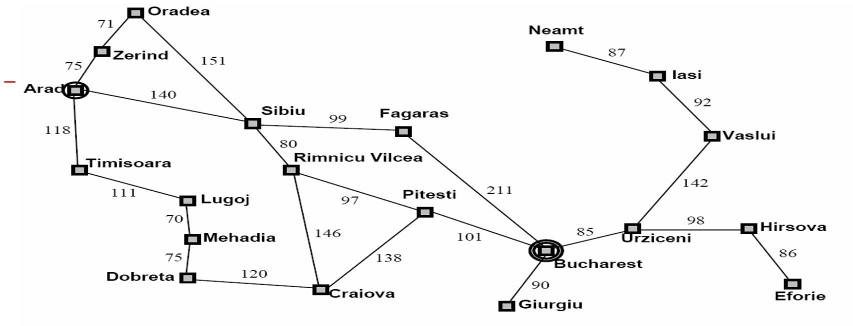
{Arad},

Solution: Arad -> Sibiu -> Fagaras -> Bucharest
Cost: 140 + 99 + 211 = 450



{Arad}

Solution: Arad -> Sibiu -> Rimnicu Vilcea -> Pitesti ->Bucharest
Cost: 140 + 80 + 97 + 101 = 418



{Arad},

Frontier is a set of paths not a set of states: cycles become an issue.

Selection Rule.

- The example shows that order states are selected from the frontier has a critical effect on the operation of the search:
  - Whether or not a solution is found
  - The cost of the solution found.
  - The time and space required by the search.

Critical Properties of Search.

- Completeness: will the search always find a solution if a solution exists?
- Optimality: will the search always find the least cost solution? (when actions have costs)
- Time complexity: what is the maximum number of nodes than can be expanded or generated?
- Space complexity: what is the maximum number of nodes that have to be stored in memory?

Uninformed Search Strategies

- These are strategies that adopt a fixed rule for selecting the next state to be expanded.
- The rule does not change irrespective of the search problem being solved.
- These strategies do not take into account any domain specific information about the particular search problem.
- Popular uninformed search techniques:
  - Breadth-First, Uniform-Cost, Depth-First, Depth-Limited, and Iterative-Deepening search

Selecting vs. Sorting

- A simple equivalence we will exploit
  - Order the elements on the frontier.
  - Always select the first element.
- Any selection rule can be achieved by employing an appropriate ordering of the frontier set.

Breadth First.

- Place the successors of the current state at the end of the frontier.

- Example:
  - let the states be the positive integers {0,1,2,…}
  - let each state n have as successors n+1 and n+2
    - E.g. S(1) = {2, 3}; S(10) = {11, 12}
  - Start state 0
  - Goal state 5

Breadth First Example.

{0<>}

Breadth First Properties

- Measuring time and space complexity.
  - let b be the maximum number of successors of any state.
  - let d be the number of actions in the shortest solution.

Breadth First Properties

- Completeness?
  - The length of the path from the initial state to the expanded state must increase monotonically.
    - we replace each expanded state with states on longer paths.
    - All shorter paths are expanded prior before any longer path.
  - Hence, eventually we must examine all paths of length d, and thus find the shortest solution.

Breadth First Properties

- Time Complexity?

- $1 + b + b^2 + b^3 + \ldots + b^{d-1} + b^d + b(b^d - 1) = O(b^{d+1})$

Breadth First Properties

- Space Complexity?
  - $O(b^{d+1})$: If goal node is last node at level d, all of the successors of the other nodes will be on the frontier when the goal node is expanded $b(b^d - 1)$
- Optimality?
  - Will find shortest length solution
    - least cost solution?

Breadth First Properties

- Space complexity is a real problem.
  - E.g., let b = 10, and say 1000 nodes can be expanded per second and each node requires 100 bytes of storage:

Depth	Nodes	Time	Memory
1	1	1 millisec.	100 bytes
6	10^6	18 mins.	111 MB
8	10^8	31 hrs.	11 GB

- Run out of space long before we run out of time in most applications.

Uniform Cost Search.

- Keep the frontier sorted in increasing cost of the path to a node.
- Always expand the least cost node.
- Identical to Breadth first if each transition has the same cost.

- Example:
  - let the states be the positive integers {0,1,2,…}
  - let each state n have as successors n+1 and n+2
  - Say that the n+1 action has cost 2, while the n+2 action has cost 3.

Uniform-Cost Search Example

{0}

Uniform-Cost Search

- Completeness?
  - If each transition has costs $\geq \varepsilon > 0$.
  - The previous argument used for breadth first search holds: the cost of the expanded state must increase monotonically.

Uniform-Cost Search

- Time and Space Complexity?
  - $O(b^{C^*/\varepsilon})$ where C* is the cost of the optimal solution.
    - Difficulty is that there may be many long paths with cost ≤ C*; Uniform-cost search must explore them all.

Uniform-Cost Search

- Optimality?
  - Finds optimal solution if each transition has cost ≥ ε > 0.
    - Explores paths in the search space in increasing order of cost. So must find minimum cost path to a goal before finding any higher costs paths.

Uniform-Cost Search. Proof of Optimality.

Lemma 1.

Let c(n) be the cost of the path to node n. If n2 is expanded after n1 then c(n1) ≤ c(n2).

Proof: there are 2 cases:

a. n2 was on the frontier when n1was expanded
b. n2 was added to the frontier when n1was expanded

Uniform-Cost Search. Proof of Optimality.

Lemma 2.

When n is expanded every path with cost strictly less than c(n) has already been expanded (i.e., every node on it has been expanded).

Proof:

Let <Start, n0, n1, …, nk> be a path with cost less than c(n). Our claim is that every node on this path must have already been expanded by the time n is expanded by uniform-cost search.

Uniform-Cost Search. Proof of Optimality.

Lemma 3.

The first time uniform-cost expands a state, it has found the minimal cost path to it (it might later find other paths to the same state but none of them can be less costly).

Proof:

Depth First Search

- Place the successors of the current state at the front of the frontier.

Depth First Search Example

(applied to the example of BFS)

Depth First Properties

- Completeness?
  - Infinite paths?
  - Prune paths with duplicate states?
- Optimality?

Depth First Properties

- Time Complexity?
  - $O(b^m)$ where m is the length of the longest path in the state space.
  - Very bad if m is much larger than d, but if there are many solution paths it can be much faster than breadth first.

Depth First Backtrack Points

- Unexplored siblings of nodes along current path.
  - These are the nodes on the frontier.

Depth First Properties

- Space Complexity?
  - O(bm), linear space!
    - Only explore a single path at a time.
    - The frontier only contains the deepest states on the current path along with the backtrack points.

Depth Limited Search

- Breadth first has computational, especially, space problems. Depth first can run off down a very long (or infinite) path.
- Depth limited search.
  - Perform depth first search but only to a pre-specified depth limit L.
  - No node on a path that is more than L steps from the initial state is placed on the Frontier.
  - We "truncate" the search by looking only at paths of length L or less.
- Now infinite length paths are not a problem.
- But will only find a solution if a solution of length $\leq$ L exists.

Depth Limited Search

```
DLS(Frontier, Sucessors, Goal?)
   If Frontier is empty return failure

   Curr = select state from Frontier

   If(Goal?(Curr)) return Curr.

   If Depth(Curr) < L
       Frontier' = (Frontier – {Curr}) U Successors(state)

   Else
       Frontier' = Frontier – {Curr}
       CutOffOccured = TRUE.

   return DLS(Frontier', Successors, Goal?)
```


Iterative Deepening Search.

- Take the idea of depth limited search one step further.
- Starting at depth limit L = 0, we iteratively increase the depth limit, performing a depth limited search for each depth limit.
- Stop if no solution is found, or if the depth limited search failed without cutting off any nodes because of the depth limit.

Iterative Deepening Search Example

Iterative Deepening Search Properties

- Completeness?
  - Yes, if a minimal length solution of length d exists. What happens when the depth limit L=d?
    What happens when the depth limit L<d?
- Time Complexity?

Iterative Deepening Search Properties

- Time Complexity
  - $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
  - E.g. b=4, d=10
    - $(11)*4^0 + 10*4^1 + 9*4^2 + \ldots + 2*4^9 = 815{,}555$
    - $4^{10} = 1{,}048{,}576$
    - Most nodes lie on bottom layer.
    - In fact IDS can be more efficient than breadth first search: nodes at limit are not expanded. BFS must expand all nodes until it expand a goal node.

Breadth first can explore more nodes than IDS.

Iterative Deepening Search Properties

- Space Complexity
  - O(bd) Still linear!
- Optimal?
  - Will find shortest length solution which is optimal if costs are uniform.
  - If costs are not uniform, we can use a "cost" bound instead.
    - Only expand paths of cost less than the cost bound.
    - Keep track of the minimum cost unexpanded path in each depth first iteration, increase the cost bound to this on the next iteration.
    - This can be very expensive. Need as many iterations of the search as there are distinct path costs.

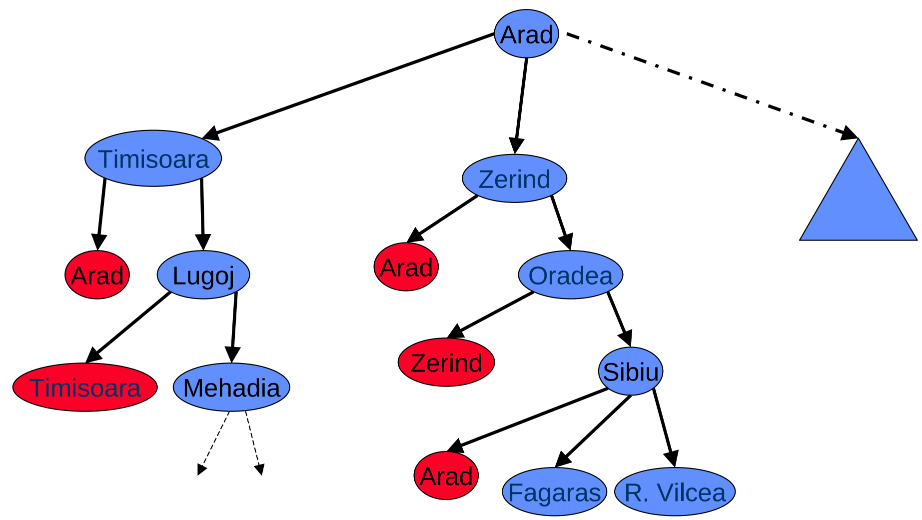
Iterative Deepening Search Properties

- Consider space with three paths of length 3, but each action having a distinct cost.

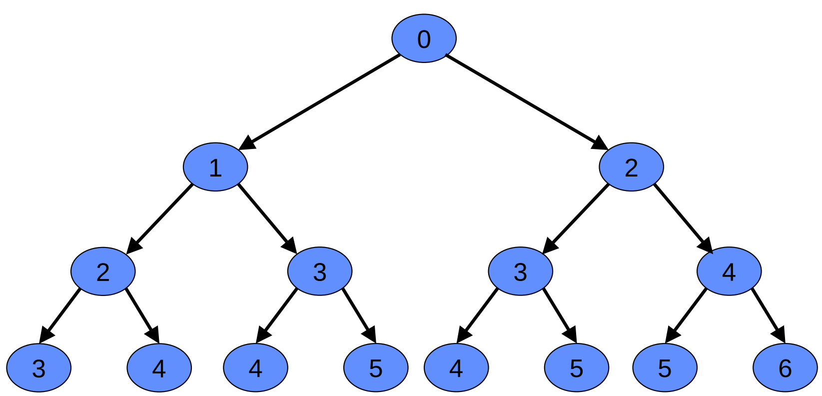
Cycle Checking

- Path checking
  - Recall paths are stored on the frontier (this allows us to output the solution path).
    - If $<S,n_1,\ldots,n_k>$ is a path to node n_k, and we expand n_k to obtain child c, we have
      $<S,n_1,\ldots,n_k,c>$
    - As the path to "c".
  - Path checking:
    - Ensure that the state c is not equal to the state reached by any ancestor of c along this path.
    - That is paths are checked in isolation!

Path Checking Example



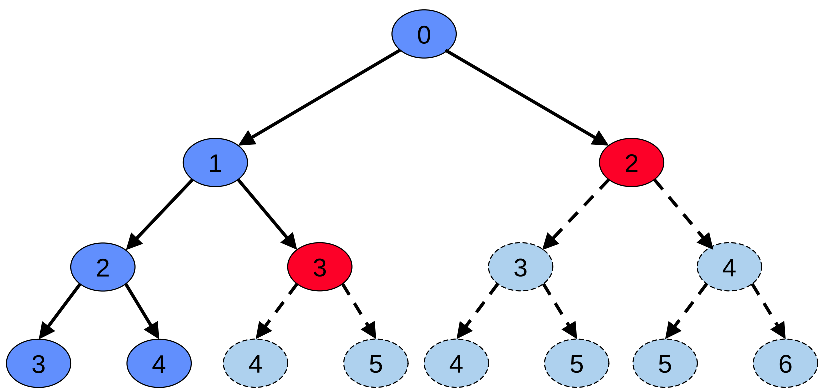
Path Checking Example



Cycle Checking

- Cycle Checking.
  - Keep track of all states previously expanded during the search.
  - When we expand n_k to obtain child c
    - ensure that c is not equal to any previously expanded state.
  - This is called cycle checking, or multiple path checking.
  - Why can't we utilize this technique with depth-first search?
    - If we modify depth-first search to do cycle checking what happens to space complexity?

Cycle Checking Example



Cycle Checking

- High space complexity, only useful with breadth first search.
- There is an additional issue when we are looking for an optimal solution
  - With uniform-cost search, we still find an optimal solution
    - The first time uniform-cost expands a state it has found the minimal cost path to it.
  - This means that the nodes rejected by cycle checking can't have better paths.
  - We will see later that we don't always have this property when we do heuristic search.

Heuristic Search.

- In uninformed search, we don't try to evaluate which of the nodes on the frontier are most promising. We never "look-ahead" to the goal.
  - E.g., in uniform cost search we always expand the cheapest path. We don't consider the cost of getting to the goal.
- Often we have some other knowledge about the merit of nodes, e.g., going the wrong direction in Romania.

Heuristic Search.

- Merit of a frontier node: different notions of merit.
  - If we are concerned about the cost of the solution, we might want a notion of merit of how costly it is to get to the goal from that search node.
  - If we are concerned about minimizing computation in search we might want a notion of ease in finding the goal from that search node.
  - We will focus on the "cost of solution" notion of merit.

Heuristic Search.

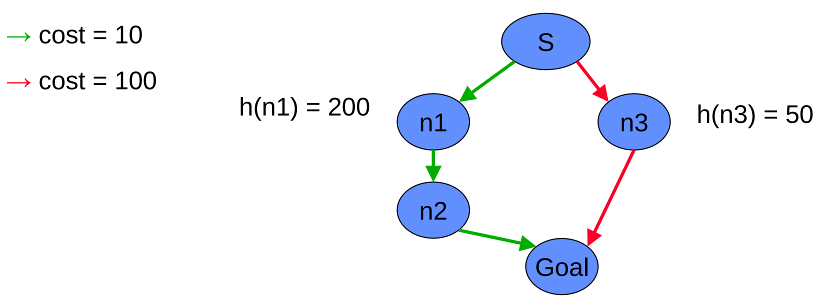
- The idea is to develop a domain specific heuristic function h(n).
- h(n) guesses the cost of getting to the goal from node n.
- There are different ways of guessing this cost in different domains. I.e., heuristics are domain specific.

Heuristic Search.

- Convention: If $h(n_1) < h(n_2)$ this means that we guess that it is cheaper to get to the goal from n_1 than from n_2.
- We require that
  - h(n) = 0 for every node n that satisfies the goal.
    - Zero cost of getting to a goal node from a goal node.

Using only h(n): Greedy best-first search.

- We use h(n) to rank the nodes on open.
  - Always expand node with lowest h-value.
- We are greedily trying to achieve a low cost solution.
- However, this method ignores the cost of getting to n, so it can be lead astray exploring nodes that cost a lot to get to but seem to be close to the goal:



A* search

- Take into account the cost of getting to the node as well as our estimate of the cost of getting to the goal from n.
- Define
  - $f(n) = g(n) + h(n)$
    - g(n) is the cost of the path to node n
    - h(n) is the heuristic estimate of the cost of getting to a goal node from n.
- Now we always expand the node with lowest f-value on the frontier.
- The f-value is an estimate of the cost of getting to the goal via this node (path).

Conditions on h(n)

- We want to analyze the behavior of the resultant search.
- Completeness, time and space, optimality?
- To obtain such results we must put some further conditions on the heuristic function h(n) and the search space.

Conditions on h(n): Admissible

- We always assume that $c(n1 \rightarrow n2) \geq \varepsilon > 0$. The cost of any transition is greater than zero and can't be arbitrarily small.
- Let h*(n) be the cost of an optimal path from n to a goal node (∞ if there is no path). Then an admissible heuristic satisfies the condition
  - $h(n) \leq h^*(n)$
    - i.e. *h* always underestimates of the true cost.
- Hence
  - h(g) = 0
  - For any goal node "g"

Consistency/monotonicity.

- Is a stronger condition than $h(n) \leq h^*(n)$.
- A monotone/consistent heuristic satisfies the triangle inequality (for all nodes n1,n2):

$$h(n1) \leq c(n1 \rightarrow n2) + h(n2)$$

- Note that there might be more than one transition (action) between n1 and n2, the inequality must hold for all of them.
- Note that monotonoicity implies admissibility. Why?

Intuition behind admissibility

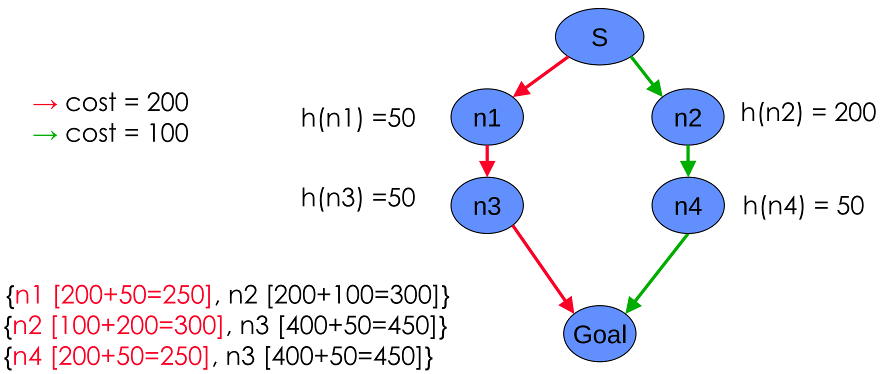
- $h(n) \leq h^*(n)$ means that the search won't miss any promising paths.
  - If it really is cheap to get to a goal via n (i.e., both g(n) and h*(n) are low), then f(n) = g(n) + h(n) will also be low, and the search won't ignore n in favor of more expensive options.
  - This can be formalized to show that admissibility implies optimality.C

Intuition behind monotonicity

- $h(n1) \le c(n1 \rightarrow n2) + h(n2)$
  - This says something similar, but in addition one won't be "locally" mislead. See next example.

Example: admissible but nonmonotonic

- The following *h* is not consistent since *h(n2)>c(n2→n4)+h(n4). But it is admissible.*



{S} → {n1 [200+50=250], n2 [200+100=300]}
→ {n2 [100+200=300], n3 [400+50=450]}
→ {n4 [200+50=250], n3 [400+50=450]}
→ {goal [300+0=300], n3 [400+50=450]}

We **do find** the optimal path as the heuristic is still admissible. **But** we are mislead into ignoring n2 until after we expand n1.

Consequences of monotonicity

1. The f-values of nodes along a path must be non-decreasing.

- Let <Start→ n1→ n2…→ nk> be a path. We claim that
  $f(ni) \le f(ni+1)$
- Proof:

$$\begin{aligned} f(ni) &= c(Start \rightarrow \ldots \rightarrow ni) + h(ni) \\ &\le c(Start \rightarrow \ldots \rightarrow ni) + c(ni \rightarrow ni+1) + h(ni+1) \\ &= c(Start \rightarrow \ldots \rightarrow ni \rightarrow ni+1) + h(ni+1) \\ &= g(ni+1) + h(ni+1) \\ &= f(ni+1). \end{aligned}$$

Consequences of monotonicity

2. If n2 is expanded after n1, then $f(n1) \le f(n2)$
   (the f-value increases monotically)

Proof:

- If n2 was on the frontier when n1 was expanded, then $f(n1) \le f(n2)$ otherwise we would have expanded n2.
- If n2 was added to the frontier after n1's expansion, then let n be an ancestor of n2 that was present when n1 was being expanded (this could be n1 itself). We have $f(n1) \le f(n)$ since A* chose n1 while n was present in the frontier. Also, since n is along the path to n2, by property (1) we have $f(n) \le f(n2)$. So, we have $f(n1) \le f(n2)$.

Consequences of monotonicity

3. When n is expanded every path with lower f-value has already been expanded.

- **Proof:** Assume by contradiction that there exists a path <Start, n0, n1, ni-1, ni, ni+1, …, nk> with $f(nk) < f(n)$ and ni is its last expanded node.
  - ni+1 must be on the frontier while n is expanded, so
    - a) by (1) $f(ni+1) \leq f(nk)$ since they lie along the same path.
    - b) since $f(nk) < f(n)$ so we have $f(ni+1) < f(n)$
    - c) by (2) $f(n) \leq f(n+1)$ because n is expanded before ni+1.
  - Contradiction from b&c!

Consequences of monotonicity

4. With a monotone heuristic, the first time A* expands a state, it has found the minimum cost path to that state.

Proof:

* Let PATH1 = <Start, n0, n1, …, nk, n> be **the first** path to n found. We have f(path1) = c(PATH1) + h(n).
* Let PATH2 = <Start, m0,m1, …, mj, n> be another path to n found later. we have f(path2) = c(PATH2) + h(n).

* By property (3), $f(path1) \leq f(path2)$

* hence: $c(PATH1) \leq c(PATH2)$

Consequences of monotonicity

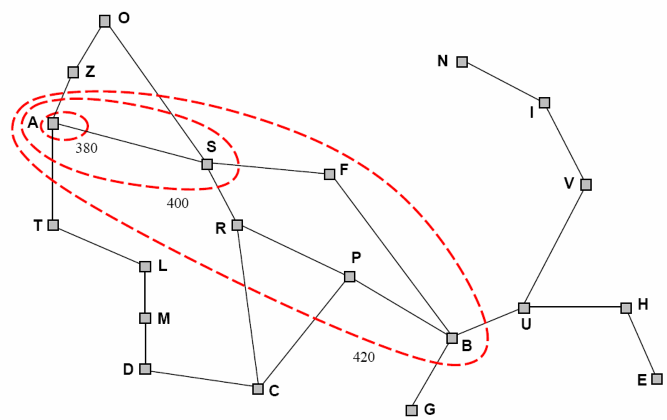
- Complete.
  - Yes, consider a least cost path to a goal node
    - SolutionPath = <Start→ n1→ …→ G> with cost c(SolutionPath)
    - Since each action has a cost $\geq \varepsilon > 0$, there are only a finite number of paths that have cost $\leq$ c(SolutionPath).
    - All of these paths must be explored before any path of cost $>$ c(SolutionPath).
    - So eventually SolutionPath, or some equal cost path to a goal must be expanded.
- Time and Space complexity.
  - When h(n) = 0, for all n h is monotone.
    - A* becomes uniform-cost search!
  - It can be shown that when $h(n) > 0$ for some n, the number of nodes expanded can be no larger than uniform-cost.
  - Hence the same bounds as uniform-cost apply. (These are worst case bounds).

Consequences of monotonicity

- Optimality
  - Yes, by (4) the first path to a goal node must be optimal.
- Cycle Checking
  - If we do cycle checking (multiple path checking) e.g. using GraphSearch instead of TreeSearch, it is still optimal. Because by property (4) we need keep only the first path to a node, rejecting all subsequent paths.

Search generated by monotonicity

Gradually adds "f-contours" of nodes (cf. breadth-first adds layers)
Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$



Admissibility without monotonicity

- When "h" is admissible but not monotonic.
  - Time and Space complexity remain the same. Completeness holds.
  - Optimality still holds (without cycle checking), but need a different argument: don't know that paths are explored in order of cost.
- Proof of optimality (without cycle checking):
  - Assume the goal path <S,…,G> found by A* has cost bigger than the optimal cost: i.e. $C^* < f(G)$.
  - There must exists a node n in the optimal path that is still in the frontier.
  - We have: $f(n)=g(n)+h(n) \leq g(n)+h^*(n)=C^* < f(G)$
  - Therefore, f(n) must have been selected before G by A*. contradiction!

Admissibility without monotonicity

- No longer guaranteed we have found an optimal path to a node ***the first time*** we visit it.
- So, cycle checking might not preserve optimality.
  - To fix this: for previously visited nodes, must remember cost of previous path. If new path is cheaper must explore again.
- contours of monotonic heuristics don't hold.

Space problem with A* (like breath-first search):

IDA* is similar to Iterative Lengthening Search: It puts the newly expanded nodes in the front of frontier! Two new parameters:

- curBound (any node with a bigger f value is discarded)
- smallestNotExplored (the smallest f value for discarded nodes in a round) when frontier becomes empty, the search starts a new round with this bound.

Building Heuristics: Relaxed Problem

- One useful technique is to consider an easier problem, and let h(n) be the cost of reaching the goal in the easier problem.
- 8-Puzzle moves.
  - Can move a tile from square A to B if
    - A is adjacent (left, right, above, below) to B
    - and B is blank
- Can relax some of these conditions
  1. can move from A to B if A is adjacent to B (ignore whether or not position is blank)
  2. can move from A to B if B is blank (ignore adjacency)
  3. can move from A to B (ignore both conditions).

Building Heuristics: Relaxed Problem

- #3 leads to the misplaced tiles heuristic.
  - To solve the puzzle, we need to move each tile into its final position.
  - Number of moves = number of misplaced tiles.
  - Clearly h(n) = number of misplaced tiles ≤ the h*(n) the cost of an optimal sequence of moves from n.
- #1 leads to the manhattan distance heuristic.
  - To solve the puzzle we need to slide each tile into its final position.
  - We can move vertically or horizontally.
  - Number of moves = sum over all of the tiles of the number of vertical and horizontal slides we need to move that tile into place.
  - Again h(n) = sum of the manhattan distances ≤ h*(n)
    - in a real solution we need to move each tile at least that that far and we can only move one tile at a time.

Building Heuristics: Relaxed Problem

- The optimal cost to nodes in the relaxed problem is an admissible heuristic for the original problem!

  Proof: the optimal solution in the original problem is a (*not necessarily optimal*) solution for relaxed problem, therefore it must be at least as expensive as the optimal solution in the relaxed problem.
- Comparison of IDS and A* (average total nodes expanded):

Depth	IDS	A*(Misplaced)	A*(Manhattan)
10	47,127	93	39
14	3,473,941	539	113
24	---	39,135	1,641

Let h1=Misplaced, h2=Manhattan

- Does h2 **always** expand less nodes than h1?
  - Yes! Note that h2 dominates h1, i.e. for all n: h1(n)≤h2(n). From this you can prove h2 is faster than h1.
  - Therefore, among several admissible heuristic the one with highest value is the fastest.

Building Heuristics: Pattern databases.

- Admissible heuristics can also be derived from solution to **subproblems: Each state is mapped into a partial specification, e.g. in 15-puzzle only *position of specific tiles matters.***

- Here are goals for two sub-problems (called Corner and Fringe) of 15puzzle. If you want to know how they came up with these subproblems? Here is the paper.
- Note that the goal state here for15-puzzle is **different** than what we have defined in Assignment1).

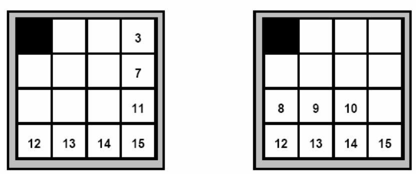


Fig. 2. The Fringe and Corner Target Patterns.

- By searching backwards from these goal states, we can compute the distance of any configuration of these tiles to their goal locations. We are ignoring the identity of the other tiles.
- For any state n, the number of moves required to get these tiles into place form a lower bound on the cost of getting to the goal from n.

Building Heuristics: Pattern databases.

- These configurations are stored in a database, along with the number of moves required to move the tiles into place.
- The maximum number of moves taken over all of the databases can be used as a heuristic.
- On the 15-puzzle
  - The fringe data base yields about a 345 fold decrease in the search tree size.
  - The corner data base yields about 437 fold decrease.
- Some times disjoint patterns can be found, then the number of moves can be added rather than taking the max.

Local Search

- So far, we keep the paths to the goal.
- For some problems (like 8-queens) we don't care about the path, we only care about the solution. Many real problem like Scheduling, IC design, and network optimizations are of this form.
- Local search algorithms operate using a single Current state and generally move to neighbors of that state.
- There is an objective function that tells the value of each state. The goal has the highest value (global maximum).
- Algorithms like Hill Climbing try to move to a neighbor with the highest value.
- Danger of being stuck in a local maximum. So some randomness is added to "shake" out of local maxima.
- Simulated Annealing: Instead of the best move, take a random move and if it improves the situation then always accept, otherwise accept with a probability <1.
- [If intrested read these two algorithms from the Book].