**Session-6**

**A\* Search**

**A\* Search Algorithm**

A\* search algorithm is a form of Best First Search algorithm.

**Best First Search (BFS):**

Best First Search is a way of combining the advantages of both depth first search and breadth first search.

At each step of the best-first search process, we select the most promising of the nodes we have generated so far. This is done by applying an appropriate heuristic function to each of them. We then expand the chosen node by using the rules to generate its successors. If one of them is solution, we quit. If not all those new nodes are added to the set of nodes generated so far. Again the most promising node is selected and the process continues.

**Algorithm:**

1. Start with OPEN containing just the initial state

2. Until a goal is found or there are no nodes left on OPEN do:

a. Pick the best node on OPEN

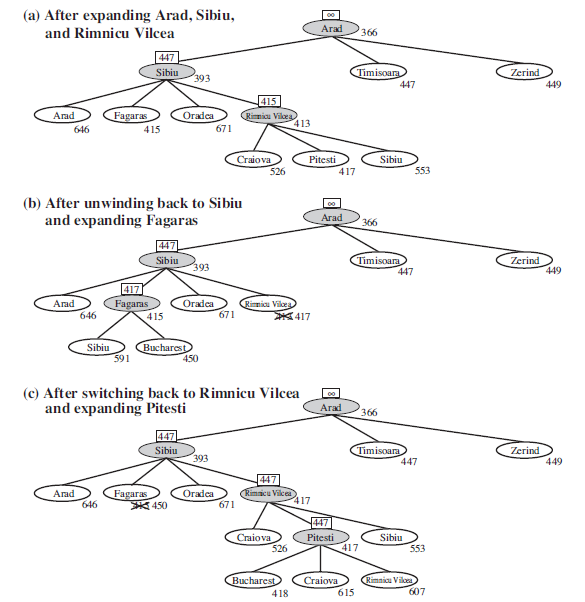
b. Generate its successors

c. For each successor do:

i. If it has not been generated before, evaluate it, add it to OPEN, and record its parent.

ii. If it has been generated before, change the parent if this new path is better than the previous one. In that case, update the cost of getting to this node and to any successors that this node may already have.

**Eg.** Apply best first search to find path from Arad to Bucharest.



**Figure:** Illustrating Best First Search

**A\* Search:**

The most widely known form of best-first search is called **A-star** **search.** It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal:

**f(n) = g(n) + h(n) .**

Since **g(n)** gives the path cost from the start node to node n, and

**h(n)** is the estimated cost of the cheapest path from n to the goal,

**f(n)** = estimated cost of the cheapest solution through n .

Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of g(n) + h(n). It turns out that this strategy is more than just reasonable: provided that the heuristic function h(n) satisfies certain conditions.

Observations:

1. If g=0 then the node that seems closest to the goal will be chosen
2. If g=1 then the path involving fewest number of steps will be chosen
3. If the cost of going from one node to another varies, then g reflects it and we find a cheaper path
4. If h=0 , the search will be controlled by g
5. If h=0 and g= 0 then the search strategy will be random.
6. A\* search is both complete and optimal. The algorithm is identical to UNIFORM-COST-SEARCH except that A\* uses g + h instead of g.

**Algorithm:**

1. Start with OPEN containing only initial node. Set that node’s g(n) value to 0, its h(n) value to whatever it is, and its f(n) value to h+0 or h(n). Set CLOSED to empty list.
2. Until a goal node is found, repeat the following procedure:

i. If there are no nodes on OPEN, report failure.

ii. Otherwise pick the node on OPEN with the lowest f(n) value. Call it BESTNODE.

iii. Remove it from OPEN. Place it in CLOSED.

iv. See if the BESTNODE is a goal state. If so exit and report a solution.

v. Otherwise, generate the successors of BESTNODE but do not set the BESTNODE to point to them yet.

1. For each of the SUCCESSOR, do the following:

**i.** Set SUCCESSOR to point back to BESTNODE. (These backwards links will make it possible to recover the path once a solution is found.)

ii. Compute g(SUCCESSOR) = g(BESTNODE) + the cost of getting from BESTNODE to SUCCESSOR

iii. See if SUCCESSOR is the same as any node on OPEN. If so call the node OLD. See whether it is cheaper to get to OLD via its current parent or SUCCESSOR via BESTNODE by comparing their g values. If SUCCESSOR is cheaper, then reset OLD’s parent to point to BESTNODE, record the new cheaper path in g(OLD) and update f’(OLD).

iv. If SUCCESSOR was not on OPEN, see if it is on CLOSED. If so, call the node on CLOSED OLD and add OLD to the list of BESTNODE’s successors.

Check to see if the new path is better. If so, set the parent link and g and f’ values appropriately.

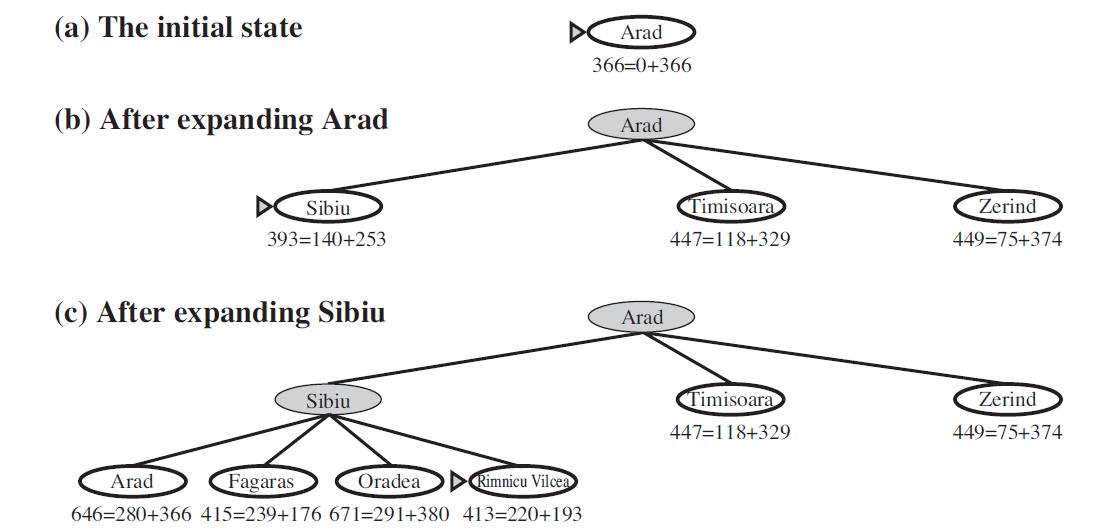
We must propagate the improvements to OLD’s successors. OLD points to successors. Each successor, in turn, points to its successors, and so forth until each branch terminates with a node that either is still on OPEN or has no successors.

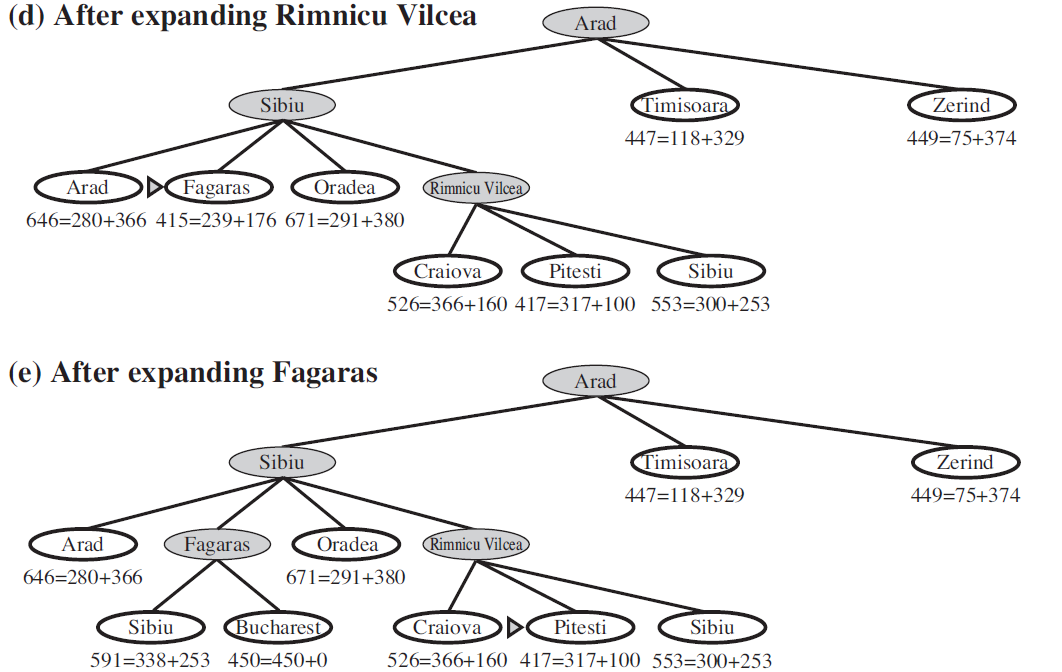
So to propagate the new cost downward, do a depth-first traversal of the tree starting at OLD, changing each node’s g value (and thus also its f’ value), terminating each branch when you reach either a node with no successor or a node to which an equivalent or better path has already been found.

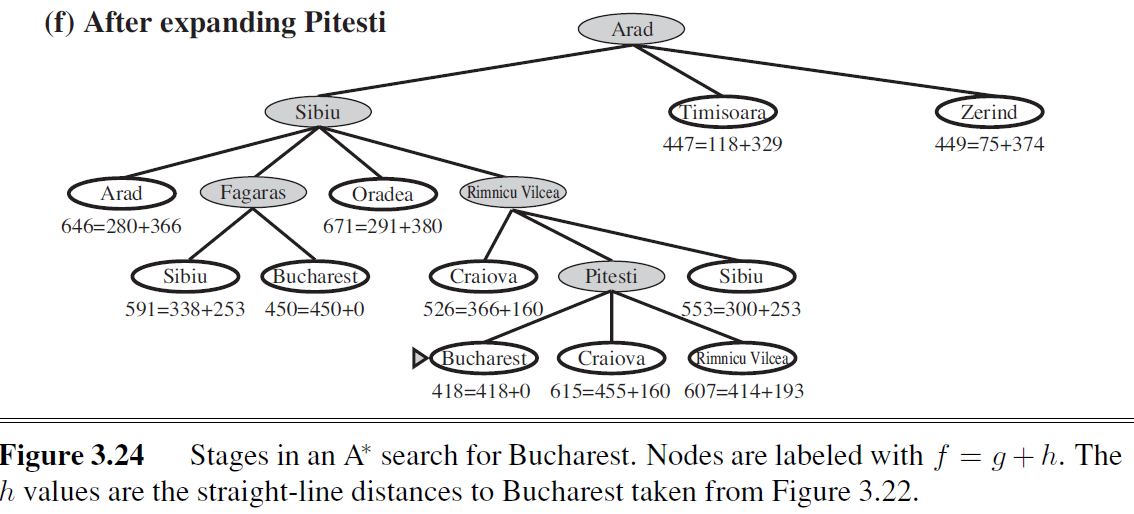
v. If SUCCESSOR was not already on either OPEN or CLOSED, then put it on OPEN and add it to the list of BESTNODE’s successors.

Compute f ’ (SUCCESSOR) = g(SUCCESSOR) + h’(SUCCESSOR)

**Eg. Apply A\* to find the path from Arad to Bucharest.**

****

****

****

For problems with constant step costs, the growth in run time as a function of the optimal solution depth d is analyzed in terms of the **absolute error** or the **relative error** of the heuristic. The absolute error is defined as

Δ ≡ h\* − h, where h\* is the actual cost of getting from the root to the goal, and the relative error is defined as ε ≡ (h\* − h)/h\*

**Advantages:**

1. A\* search algorithm is the best algorithm than other search algorithms.
2. A\* search algorithm is optimal and complete.
3. This algorithm can solve very complex problems.

**Disadvantages:**

1. It does not always produce the shortest path as it mostly based on heuristics and approximation.
2. The main drawback of A\* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

**Performance of A\* Search:**

**Complete:** A\* algorithm is complete as long as Branching factor is finite and Cost at every action is fixed.

**Optimal:** A\* search algorithm is optimal if it follows below two conditions:

**Admissible:** the first condition requires for optimality is that h(n) should be an admissible heuristic for A\* tree search. An admissible heuristic is optimistic in nature.

**Consistency:** Second required condition is consistency for only A\* graph-search.

If the heuristic function is admissible, then A\* tree search will always find the least cost path.

**Time Complexity:** The time complexity of A\* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time complexity is O(bd), where b is the branching factor.

**Space Complexity:** The space complexity of A\* search algorithm is **O(bd)**

**Memory-Bounded Heuristic Search(Self-Learning)**

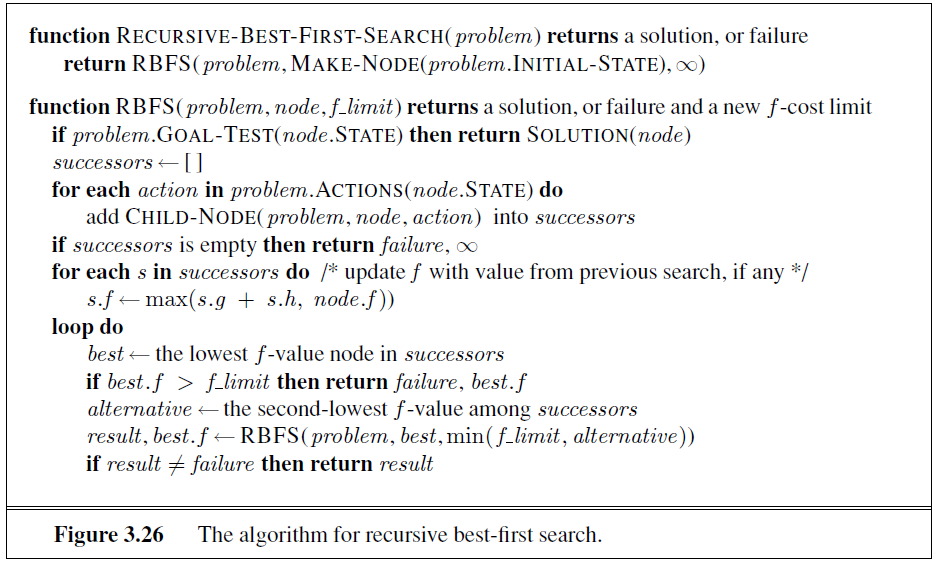
The simplest way to reduce memory requirements for A\* is to adapt the idea of Iterative Depening deepening to the heuristic search context, resulting in the iterative-deepening A\* (IDA\*) algorithm.

The main difference between IDA\* and standard iterative deepening is that the cutoff used is the *f-cost (g + h) rather than the depth; at each iteration, the cutoff value is the small*est f-cost of any node that exceeded the cutoff on the previous iteration.

IDA\* is practical for many problems with unit step costs and avoids the substantial overhead associated with keeping a sorted queue of nodes.

**Two memory bound algorithms are RBFS and MA\***

Similar to the standard best-first search, but using only linear space. Its structure is similar to that of a recursive depth-first search, but rather than continuing indefinitely down the current path, it uses the *f-limit variable to keep track of the f -value of the best alternative path available from any ancestor of the current node. If the current node exceeds this limit, the recursion unwinds back to the alternative* path. As the recursion unwinds, RBFS replaces the f-value of each node along the path with a backed-up value—the best f-value of its children. In this way, RBFS remembers the f-value of the best leaf in the forgotten sub-tree and can therefore decide whether it's worth re-expanding the sub-tree at some time later.

****

1. RBFS is an optimal algorithm if the heuristic function h(n) is admissible.
2. Its space complexity is linear in the depth of the deepest optimal solution, but its time complexity is rather difficult to characterize: it depends both on the accuracy of the heuristic function and on how often the best path changes as nodes are expanded.
3. IDA\* and RBFS suffer from using too little memory. Between iterations, IDA\* retains only a single number: the current f-cost limit. RBFS retains more information in memory, but it uses only linear space: even if more memory were available, RBFS has no way to make use of it. Because they forget most of what they have done, both algorithms may end up re-expanding the same states many times over.
4. They suffer the potentially exponential increase in complexity associated with redundant paths in graphs.

To use all available memory, can use two algorithms: - MA\* (memory-bounded A\*) and SMA\* (simplified MA\*).

SMA proceeds just like A\*, expanding the best leaf until memory is full. At this point, it cannot add a new node to the search tree without dropping an old one.

SMA\* always drops the worst leaf node—the one with the highest f -value. Like RBFS, SMA\* then backs up the value of the forgotten node to its parent. In this way, the ancestor of a forgotten sub-tree knows the quality of the best path in that sub-tree.

With this information, SMA\* regenerates the sub-tree only when all other paths have been shown to look worse than the path it has forgotten. Another way of saying this is that, if all the descendants of a node n are forgotten, then we will not know which way to go from n, but we will still have an idea of how worthwhile it is to go anywhere from n.

SMA\* expands the Best leaf and deletes the worst leaf. To avoid selecting the same node for deletion and expansion, SMA\* expands the newest best leaf and deletes the oldest worst leaf.

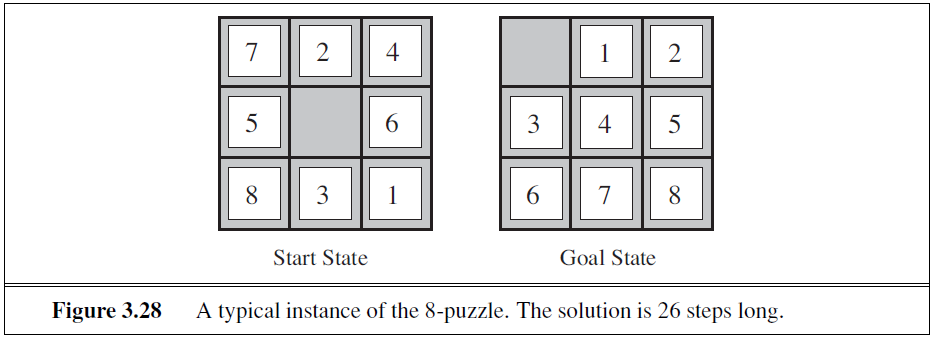
These coincide when there is only one leaf, but in that case, the current search tree must be a single path from root to leaf that fills all of memory. If the leaf is not a goal node, then even if it is on an optimal solution path, that solution is not reachable with the available memory. Therefore, the node can be discarded exactly as if it had no successors.

SMA" is complete if there is any reachable solution—that is, if d, the depth of the shallowest goal node, is less than the memory size (expressed in nodes). It is optimal if any optimal solution is reachable; otherwise, it returns the best reachable solution.

In practical terms, SMA\* is a fairly robust choice for finding optimal solutions, particularly when the state space is a graph, step costs are not uniform, and node generation is expensive compared to the overhead of maintaining the frontier and the explored set.

**HEURISTIC FUNCTIONS**

The 8-puzzle was one of the earliest heuristic search problems.



The average solution cost for a randomly generated 8-puzzle instance is about 22 steps. The branching factor is about 3. (When the empty tile is in the middle, four moves are possible; when it is in a corner, two; and when it is along an edge, three.) This means that an exhaustive tree search to depth 22 would look at about 322 ≈ 3.1×1010 states. A graph search would cut this down by a factor of about 170,000 because only 9!/2 = 181,440 distinct states are reachable. This is a manageable number, but the corresponding number for the 15-puzzle is roughly 1013, so the next order of business is to find a good heuristic function. If we want to find the shortest solutions by using A∗, we need a heuristic function that never overestimates the number of steps to the goal. There is a long history of such heuristics for the 15-puzzle; here are two commonly used candidates:

**h1** = the number of misplaced tiles. For Figure 3.28, all of the eight tiles are out of position, so the start state would have h1 = 8. h1 is an admissible heuristic because it is clear that any tile that is out of place must be moved at least once.

**h2** = the sum of the distances of the tiles from their goal positions. Because tiles cannot move along diagonals, the distance we will count is the sum of the horizontal and vertical distances. This is sometimes called the city block distance or Manhattan distance. h2 is also admissible because all any move can do is move one tile one step closer to the goal. Tiles 1 to 8 in the start state give a Manhattan distance of h2 = 3+1 + 2 + 2+ 2 + 3+ 3 + 2 = 18.

As expected, neither of these overestimates the true solution cost, which is 26.

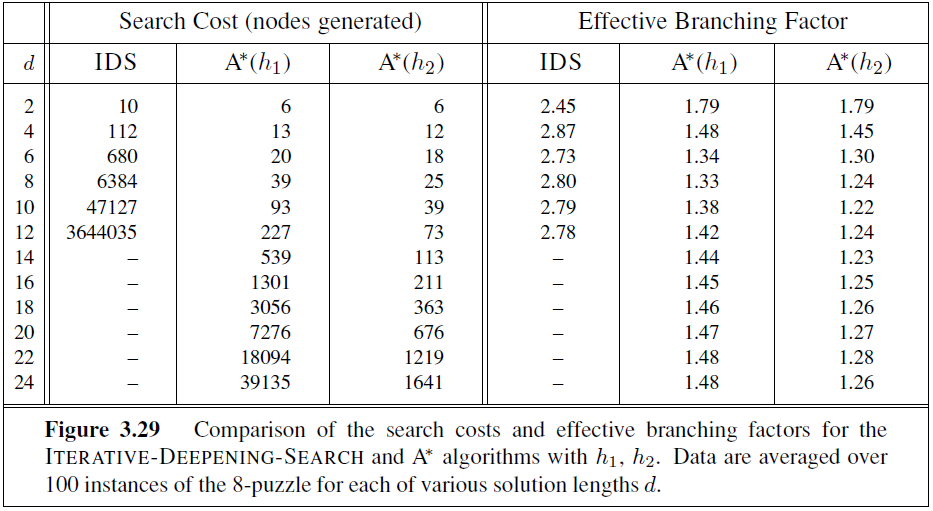
**The effect of heuristic accuracy on performance**

One way to characterize the quality of a heuristic is the **effective branching factor** b\* . If the total number of nodes generated by A\* for a particular problem is N and the solution depth is d, then b\* is the branching factor that a uniform tree of depth d would have to have in order to contain N + 1 nodes. Thus,

N + 1 = 1+b\* + (b\*)2 + ・ ・ ・ + (b\*)d

For example, if A\* finds a solution at depth 5 using 52 nodes, then the effective branching factor is 1.92.

To test the heuristic functions h1 and h2, we generated 1200 random problems with solution lengths from 2 to 24 (100 for each even number) and solved them with iterative deepening search and with A\* tree search using both h1 and h2. Figure 3.29 gives the average number of nodes generated by each strategy and the effective branching factor. The results suggest that h2 is better than h1, and is far better than using iterative deepening search. Even for small problems with d=12, A\* with h2 is 50,000 times more efficient than uninformed iterative deepening search.



It is easy to see from the definitions of the two heuristics that, for any node n, h2(n) ≥ h1(n). We thus say that h2 **dominates** h1. Domination translates directly into efficiency: A\* using h2 will never expand more nodes than A\* using h1 (except possibly for some nodes with f(n)=C\*). It is generally better to use a heuristic function with higher values, provided it is consistent and that the computation time for the heuristic is not too long.

**Generating admissible heuristics from relaxed problems:**

We have seen that both h1 (misplaced tiles) and h2 (Manhattan distance) are fairly good heuristics for the 8-puzzle and that h2 is better. How might one have come up with h2? Is it

possible for a computer to invent such a heuristic mechanically?

h1 and h2 are estimates of the remaining path length for the 8-puzzle, but they are also perfectly accurate path lengths for *simplified* versions of the puzzle. If the rules of the puzzle were changed so that a tile could move anywhere instead of just to the adjacent empty square, then h1 would give the exact number of steps in the shortest solution. Similarly, if a tile could move one square in any direction, even onto an occupied square, then h2 would give the exact number of steps in the shortest solution. A problem with fewer restrictions on the actions is called a **relaxed problem**. The state-space graph of the relaxed problem is a *supergraph* of the original state space because the removal of restrictions creates added edges in the graph.

If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically. For example, if the 8-puzzle actions are described as A tile can move from square A to square B if A is horizontally or vertically adjacent to B **and** B is blank we can generate three relaxed problems by removing one or both of the conditions:

(a) A tile can move from square A to square B if A is adjacent to B.

(b) A tile can move from square A to square B if B is blank.

(c) A tile can move from square A to square B.

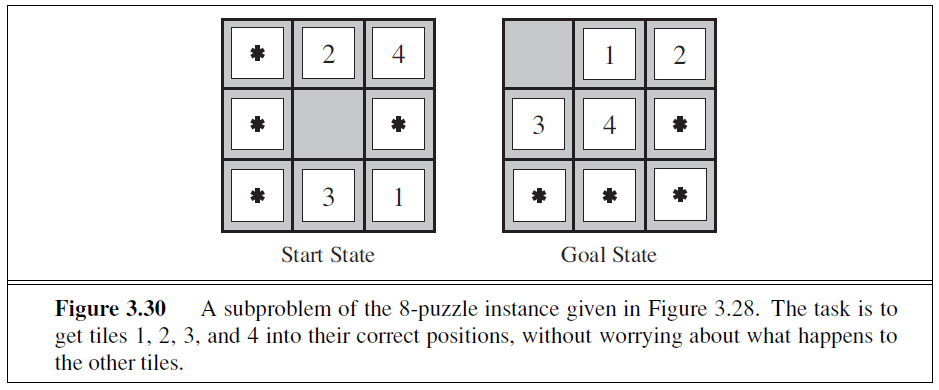
One problem with generating new heuristic functions is that one often fails to get a single “clearly best” heuristic. If a collection of admissible heuristics h1 . . .hm is available for a problem and none of them dominates any of the others, which should we choose? As it turns out, we need not make a choice. We can have the best of all worlds, by defining

h(n) = max{h1(n), . . . , hm(n)}

This composite heuristic uses whichever function is most accurate on the node in question. Because the component heuristics are admissible, h is admissible; it is also easy to prove that h is consistent. Furthermore, h dominates all of its component heuristics.

**Generating admissible heuristics from subproblems: Pattern databases**

Admissible heuristics can also be derived from the solution cost of a **subproblem** of a given problem. For example, Figure 3.30 shows a subproblem of the 8-puzzle instance in Figure3.28. The subproblem involves getting tiles 1, 2, 3, 4 into their correct positions. Clearly, the cost of the optimal solution of this subproblem is a lower bound on the cost of the complete problem. It turns out to be more accurate than Manhattan distance in some cases.



The idea behind **pattern databases** is to store these exact solution costs for every possible subproblem instance—in our example, every possible configuration of the four tiles and the blank. (The locations of the other four tiles are irrelevant for the purposes of solving the subproblem, but moves of those tiles do count toward the cost.) Then we compute an admissible heuristic hDB for each complete state encountered during a search simply by looking up the corresponding subproblem configuration in the database. The database itself is constructed by searching back13 from the goal and recording the cost of each new pattern encountered; the expense of this search is amortized over many subsequent problem instances.

The choice of 1-2-3-4 is fairly arbitrary; we could also construct databases for 5-6-7-8, for 2-4-6-8, and so on. Each database yields an admissible heuristic, and these heuristics can be combined, as explained earlier, by taking the maximum value. A combined heuristic of this kind is much more accurate than the Manhattan distance; the number of nodes generated when solving random 15-puzzles can be reduced by a factor of 1000. One might wonder whether the heuristics obtained from the 1-2-3-4 database and the 5-6-7-8 could be *added*, since the two subproblems seem not to overlap. Would this still give an admissible heuristic? The answer is no, because the solutions of the 1-2-3-4 subproblem and the 5-6-7-8 subproblem for a given state will almost certainly share some moves—it is unlikely that 1-2-3-4 can be moved into place without touching 5-6-7-8, and vice versa. But what if we don’t count those moves? That is, we record not the total cost of solving the 1-2-3-4 subproblem, but just the number of moves involving 1-2-3-4. Then it is easy to see that the sum of the two costs is still a lower bound on the cost of solving the entire problem. This is the idea behind **disjoint pattern databases**. With such databases, it is possible to solve random 15-puzzles in a few milliseconds—the number of nodes generated is reduced by a factor of 10,000 compared with the use of Manhattan distance. For 24-puzzles, a speedup of roughly a factor of a million can be obtained.

Disjoint pattern databases work for sliding-tile puzzles because the problem can be divided up in such a way that each move affects only one subproblem—because only one tile is moved at a time. For a problem such as Rubik’s Cube, this kind of subdivision is difficult because each move affects 8 or 9 of the 26 cubies. More general ways of defining additive,

admissible heuristics have been proposed that do apply to Rubik’s cube but they have not yielded a heuristic better than the best non additive heuristic for the problem.

**Learning heuristics from experience**

A heuristic function h(n) is supposed to estimate the cost of a solution beginning from the

state at node n. One solution was given in the preceding sections—namely, to devise relaxed problems for which an optimal solution can be found easily. Another solution is to learn from experience. “Experience” here means solving lots of 8-puzzles.

For instance, Each optimal solution to an 8-puzzle problem provides examples from which h(n) can be learned. Each example consists of a state from the solution path and the actual cost of the solution from that point. From these examples, a learning algorithm can be used to construct a function h(n) that can (with luck) predict solution costs for other states that arise during search.

Inductive learning methods work best when supplied with features of a state that are relevant to predicting the state’s value, rather than with just the raw state description. For example, the feature “number of misplaced tiles” might be helpful in predicting the actual distance of a state from the goal. Let’s call this feature x1(n). We could take 100 randomly generated 8-puzzle configurations and gather statistics on their actual solution costs. We might find that when x1(n) is 5, the average solution cost is around 14, and so on. Given these data, the value of x1 can be used to predict h(n). Of course, we can use several features. A second feature x2(n) might be “number of pairs of adjacent tiles that are not adjacent in the goal state.” How should x1(n) and x2(n) be combined to predict h(n)?

A common approach is to use a linear combination: h(n) = c1x1(n) + c2x2(n) . The constants c1 and c2 are adjusted to give the best fit to the actual data on solution costs. One expects both c1 and c2 to be positive because misplaced tiles and incorrect adjacent pairs

make the problem harder to solve. Notice that this heuristic does satisfy the condition that

h(n)=0 for goal states, but it is not necessarily admissible or consistent.

**Summary:**

This **CO1** introduced methods that an agent can use to select actions in environments that are deterministic, observable, static, and completely known. In such cases, the agent can construct sequences of actions that achieve its goals; this process is called **search**.

• Before an agent can start searching for solutions, a **goal** must be identified and a well defined **problem** must be formulated.

• A problem consists of five parts: the **initial state**, a set of **actions**, a **transition model** describing the results of those actions, a **goal test** function, and a **path cost** function. The environment of the problem is represented by a **state space**. A **path** through the state space from the initial state to a goal state is a **solution**.

• Search algorithms treat states and actions as **atomic**: they do not consider any internal structure they might possess.

• A general TREE-SEARCH algorithm considers all possible paths to find a solution, whereas a GRAPH-SEARCH algorithm avoids consideration of redundant paths.

• Search algorithms are judged on the basis of **completeness**, **optimality**, **time complexity**,

and **space complexity**. Complexity depends on b, the branching factor in the state space, and d, the depth of the shallowest solution.

• **Uninformed search** methods have access only to the problem definition. The basic

algorithms are as follows:

**– Breadth-first search** expands the shallowest nodes first; it is complete, optimal for unit step costs, but has exponential space complexity.

**– Uniform-cost search** expands the node with lowest path cost, g(n), and is optimal

for general step costs.

**– Depth-first search** expands the deepest unexpanded node first. It is neither complete nor optimal, but has linear space complexity. **Depth-limited search** adds a

depth bound.

**– Iterative deepening search** calls depth-first search with increasing depth limits until a goal is found. It is complete, optimal for unit step costs, has time complexity

comparable to breadth-first search, and has linear space complexity.

**– Bidirectional search** can enormously reduce time complexity, but it is not always

applicable and may require too much space.

• **Informed search** methods may have access to a **heuristic** function h(n) that estimates the cost of a solution from n.

**–** The generic **best-first search** algorithm selects a node for expansion according to

an **evaluation function**.

**– Greedy best-first search** expands nodes with minimal h(n). It is not optimal but is often efficient

**– A**\* **search** expands nodes with minimal f(n) = g(n) + h(n). A\* is complete and optimal, provided that h(n) is admissible (for TREE-SEARCH) or consistent (for GRAPH-SEARCH). The space complexity of A\* is still prohibitive.

**– RBFS** (recursive best-first search) and **SMA**\* (simplified memory-bounded A\*) are robust, optimal search algorithms that use limited amounts of memory; given enough time, they can solve problems that A\* cannot solve because it runs out of memory.

• The performance of heuristic search algorithms depends on the quality of the heuristic function. One can sometimes construct good heuristics by relaxing the problem definition, by storing pre-computed solution costs for sub-problems in a pattern database, or by learning from experience with the problem class.