Essentials of the A* Algorithm

Keith L. Downing

Search problems are ubiquitous in Artificial Intelligence (AI), and they are often so complex that standard brute-force methods (i.e. those that try every possibility) are infeasible to the point of requiring many centuries of computer time to test all potential solutions (i.e. states in the search space).

AI search methods try to exploit *knowledge* of the problem in order to make *intelligent* moves in search space. The core of this knowledge consists of a rough estimate of the distance from a given state to the goal. This is known as *heuristic* information, computed by a *heuristic function* that takes a search state as input and produces a distance-to-goal measure.

Search that exploits heuristics is often called *best-first search*. It is typically applied to search spaces in which solutions are gradually pieced together by applying operators (from a finite collection) to existing states, thereby *expanding* those states - hopefully in a direction toward the goal. It contrasts with *depth-first* and *breadth-first* search, which ignore *nearness-to-goal* estimates and simply expand states based on a pre-defined, knowledge-free, protocol. In depth-first search, a host of operators are applied to a single state, in series, until either a solution or failure state is reached. In breadth-first search, all immediate successors of a single state (i.e. those achievable by one operator application) are generated before more distant neighbors are considered.

In terms of search trees, breadth-first search involves a methodical downward expansion of the tree, with each horizontal level produced before the next. Conversely, depth-first search begins with the expansion of a single vine from the root to a leaf, followed by repeated plunges to leaf levels from both root and intermediate nodes.

Breadth-first search has the advantage of never overlooking a state, due to its methodical *sweep* of search space, but it suffers the drawback of taking a long time to get to solutions, since these typically reside at significant depths of the tree. Depth-first search can quickly find a solution, sometimes, but it can also waste a lot of time exploring false leads.

Best-first search combines the advantages of depth- and breadth-first variants by permitting either type of expansion pattern, depending upon heuristic information (along with exact information about the distance from the root to the current state). Thus, a best-first search might plunge halfway to the leaf before realizing that the current avenue of exploration (i.e. vine) is less promising than an unexpanded node higher up in the tree. At a later date, it might return to the vine as other states reveal themselves to be less-than-promised. At any rate, a best-first search is not a simple series of plunges or level expansions, but a heuristic-based movement along the *horizon* of the tree, i.e., the set of unexpanded states, in search of promising nodes. Figure 1 illustrates the different expansion patterns for the three search types.

The classic best-first search procedure is the *A* algorithm*, designed by Hart in 1968. The heart of A* is the following equation for evaluating a state, s, in the search space:

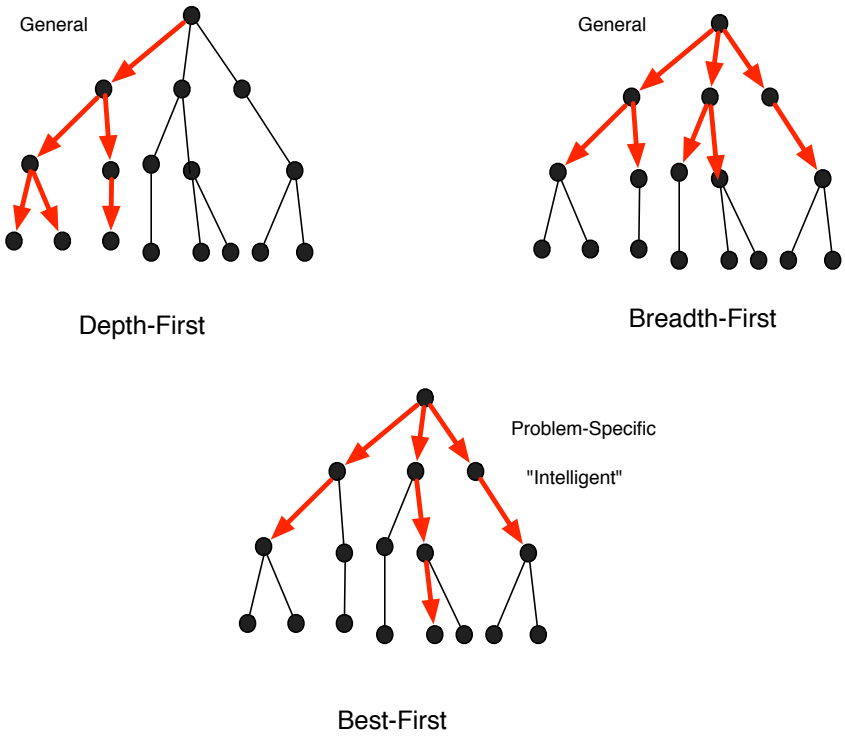

Figure 1: Comparison of 3 common tree-based search techniques: depth-first, breadth-first, and best-first. Thin lines denote unexplored paths in search space, while thick arrows represent explored areas. The children of each node represent states reachable via one operation applied to the parent state.

$$f(s) = g(s) + h(s) \tag{1}$$

Here, g(s) denotes the cost or distance of getting from the root of the search tree to s, while h(s) represents the heuristic: an estimate of the distance from s to a goal state. Thus, f(s) represents the total expected cost of a solution path going from the root, through s, to a goal.

A* applies well to problems in which:

1. An initial state, s_0 is given.
2. There exists a well-defined, finite set of operators, π, for mapping states into other states.
3. A goal state, s_g can be described, either explicitly (i.e., *it looks just like* **this**) or implicitly (i.e., *it has* **these** *properties*).
4. A *solution* consists of an entire **sequence** of states and actions from s_0 to s_g, not just the goal itself.
5. The problem requires a minimum-cost solution, which usually implies that as few operators as possible should be used. In cases where operators have varying costs, the emphasis shifts to exploiting the lower-cost operators when possible.

Consider the checkers example of Figure 2, which is easy to explain but reasonably hard to solve. Here, the initial state, s_0 is shown in the figure, while s_g is explained in the caption. The operators are simple swaps of checkers. Since the problem statement asks for a least-move solution, each swap has the same cost, which, for simplicity, we assume to be 1. Hence, for any state, s, g(s) is simply the number of moves used to get from s_0 to s.

7	24	10	19	3
12	20	8	22	23
2	15	25	18	13
11	21	5	9	16
17	4	14	1	6

Figure 2: The 25-Checkers Game: In the least number of moves, convert the pattern of checkers above into a sequential ordering, where checkers 1-5 line the top row in order (1 in the left corner, 2 on it's right, etc.), 6-10 line the second row, etc. A *move* consists of a simple vertical or horizontal swap of two adjacent checkers.

Importantly, A* takes account of the fact that several paths may lead from s_0 to s, so g(s) is always recognized as an estimate that can be improved. To wit, A* allows child states (nodes) to have several parents, with

the parent on the shortest path to the root being the focal one. Thus, A* supports search in graphs, not just trees.

The heuristic function, h(s), for the checkers example must compute an expected distance to the goal in terms of the number of remaining swaps. Such estimates are notoriously difficult, but a standard approach that works reasonably well on many problems is to calculate the distance from each checker to its location in the goal formation. The sum of these distances, D, is then the basis for h(s).

In this case, since each swap moves two checkers, $\frac{D}{2}$ might be a better estimate. Another factor is the distance metric used to calculate D. Although it may be the straight-line Euclidian distance between two points on the checker board, this can easily underestimate distance-to-goal, since diagonal movements are illegal in the game but implicitly permitted by the heuristic. For example, the Euclidean distance between the bottom right and upper left corners is only $4\sqrt{2} \approx 5.6$, or even just 4 if we consider each diagonal move as 1. But to move checker 6 between those two corners would require 8 swaps (4 vertical and 4 horizontal).

Consequently, in problems with restrictions against diagonal movement, the *Manhattan distance* is often employed. It is defined as the sum of the horizontal and vertical distances between points a and b. So the Manhattan distance between checkers 6 and 7 is 8, whereas that between checkers 17 and 9 is 4.

1 The Basic A* Algorithm

The pseudocode below summarizes the main activities of A*. Several aspects are further elaborated later in this document.

In A*, search states are typically encapsulated within search nodes, and the algorithm works with two lists of nodes: OPEN and CLOSED. The former is sorted by ascending f values of nodes such that nodes with a lot of promise (hence low f values) are popped from OPEN earlier. CLOSED requires no particular ordering, since once a node is added to CLOSED, it is never removed. CLOSED houses nodes that have already been expanded, while OPEN contains unexpanded nodes.

The algorithm begins by pushing a node representing the initial problem state onto the OPEN list. Then begins the core of A*, the **agenda loop**, in which nodes are popped from OPEN and expanded, producing child nodes that may be added to OPEN.

Early in the loop, failure and success are recognized and handled by returning a fail message or the solution node, respectively. Assuming neither occurs, the top node (X) from OPEN is popped, moved to CLOSED, and used to generate successor (child) nodes.

As shown in the agenda loop pseudocode, a newly-generated successor node (S) is checked for uniqueness. If it has already been generated, then the corresponding node is fetched from the OPEN or CLOSED list (since it must reside on one or the other), and the pre-existing node, S* is used in further calculations. These may involve updates to the g (and hence f) values of S*, in cases where X represents a better parent (due to a lower g value). If S* was on the CLOSED list (meaning that it probably has child nodes), then those improvements need to be passed on to all descendants (via the call to **propagate-path-improvements**).

If the node represents a unique new state, then it is inserted into OPEN at the appropriate location.

Regardless of whether S is unique or not, it (or S*) is added to the kids list of X, since the connection between X and S (S*) is important new information that could play a role in the final solution.

DEFINE **best-first-search**()

1. CLOSED $\leftarrow \emptyset$; OPEN $\leftarrow \emptyset$
2. Generate the initial node, n0, for the start state.
3. g(n0) $\leftarrow$ 0; Calculate h(n0)
4. f(n0) $\leftarrow$ g(n0) + h(n0);
5. Push n0 onto OPEN.
6. **Agenda Loop:** While no solution found do:
   - If OPEN = $\emptyset$ return FAIL
   - X $\leftarrow$ pop(OPEN)
   - push(X,CLOSED)
   - If X is a solution, return (X, SUCCEED)
   - SUCC $\leftarrow$ generate-all-successors(X)
   - For each S $\in$ SUCC do:
     - If node S* has previously been created, and if state(S*) = state(S), then S $\leftarrow$ S*.
     - push(S,kids(X))
     - If not(S $\in$ OPEN) and not(S $\in$ CLOSED) do:
       * **attach-and-eval**(S,X)
       * insert(S,OPEN) ;; OPEN is sorted by ascending f value.
     - else if g(X) + arc-cost(X,S) < g(S) then (found cheaper path to S):
       * **attach-and-eval**(S,X)
       * If S $\in$ CLOSED then **propagate-path-improvements**(S)

The **attach-and-eval** routine simply attaches a child node to a node that is now considered its best parent (so far). The child's value of g is then computed based on the parent's value plus the cost of moving from P to C (i.e., the arc cost). The heuristic value of C is assessed independently of P, and then f(C) is updated.

DEFINE **attach-and-eval**(C,P)

1. parent(C) $\leftarrow$ P
2. g(C) = g(P) + arc-cost(P,C)
3. Compute h(C)
4. f(C) $\leftarrow$ g(C) + h(C)

The propagation of path improvements recurses through the children and possibly many other descendants. Some children may not have had P as their best parent. If the updates to g(P) do not make P the best parent for a given child, then the propagation ceases along that child's portion of the search graph. However, if any child can improve its own g value due to the change in g(P), then that child will have P as its best parent and must propagate the improvement in g further, to its own children. This insures that all nodes in the search graph are always aware of their current best parent and their current best g value, given the information that has been uncovered by the search algorithm up to that point in time. The propagation of

these updates can eventually affect nodes on OPEN, thus influencing the order in which nodes get popped, and hence the direction of search.

DEFINE **propagate-path-improvements**(P)

1. For each C $\in$ kids(P) do:
   - If g(P) + arc-cost(P,C) $<$ g(C) then:
     - parent(C) $\leftarrow$ P
     - g(C) = g(P) + arc-cost(P,C)
     - f(C) $\leftarrow$ g(C) + h(C)
     - **propagate-path-improvement**(C)

2 Additional Details

The above pseudocode provides the essence of the A* algorithm. This section elaborates a few of the important and/or difficult aspects of the algorithm.

2.1 Nodes in A* Search

To implement the A* algorithm, it is useful to encapsulate search states within nodes in the search tree/graph. The class **search-node** in Figure 3 uses properties such as *parent*, *kids* to keep track of neighboring nodes in the graph, while the properties f, g, and h are key node variables in A*. The *parent* slot always denotes the *best parent found so far*. Anytime the g value of a node is updated, the f value should change accordingly, and if the node has children, their g and f values may change as well.

CLASS **search-node**

- state - an object describing a state of the search process
- g - cost of getting to this node
- h - estimated cost to goal
- f - estimated total cost of a solution path going through this node; f = g + h
- status - *open* or *closed*
- parent - pointer to best parent node
- kids - list of all successor nodes, whether or not this node is currently their best parent.

Figure 3: Definition of the search-node class.

2.2 Search States

The *state* property of a search node is a representation of a particular configuration of the problem being solved, e.g. the locations of all pieces in an 8-puzzle or of all vehicles in a Rush-Hour puzzle. For A* to

work properly and efficiently, it is important that each state has a unique identifier (e.g. a large integer) that concisely encodes the essential features of the state such that:

$$\forall s_1, s_2 \in S : id(s_1) = id(s_2) \Leftrightarrow s_1 = s_2 \tag{2}$$

where S is the set of all states in the search space. A* should maintain a hash table housing all states that it has generated, with each state's id used as its hash key. Then, when a state is generated as the successor of another, A* can quickly determine whether or not the *new* state has already been generated. Failure to recognize previously-generated states can lead to inefficiency of A* at best, and infinite loops in the worst case. Properties of the problem domain and the successor-generating function determine the potential for infinite looping; any programmer who wishes to avoid uniqueness testing should be keenly aware of the dangers or lack thereof.

2.3 The Agenda

At any point during a search, a certain set, S, of states have been generated and evaluated, i.e. have had their f values computed. Some of these states have been *expanded*, meaning that all operators that apply to them have been invoked, yielding one child state per operator. Prior to expansion, nodes are considered *open*, but their status changes to *closed* after expansion.

The key decision in best-first search is which open node in S to next expand. In the standard A* algorithm, that node is the one with the lowest f value. To implement this, A* keeps all open nodes in an *agenda* data structure, sorted by ascending f value. This is also known as the *open list*, *fringe* or *horizon* of a search process. It is usually implemented as a list or array, with new elements added via a merge sort or simple insertion. When a node is expanded, all of its children are evaluated and merged into this agenda. Very promising children will possess low f values and tend to be expanded soon after their parents, thus giving a more depth-first behavior to A*. Conversely, poor children with their high f values will end up near the end of the agenda.

Figure 4 depicts a simple scenario in which the current state of the search involves 5 nodes, two closed and three open. The expansion of node D results in 3 new open nodes with various f, g and h values.

At the highest level, best-first search is characterized by an agenda loop in which the most-promising, unexpanded node is popped from the agenda and expanded. All of the newly generated children are checked for uniqueness and, when unique, evaluated (i.e. their f values are computed) and added to the agenda.

Search continues until either a) there are no more nodes left to expand, b) a solution is found, or c) a maximum number of nodes have been expanded (a case which is not shown in the interest of brevity). If a solution emerges, best-parent pointers can be traced to connect it to the root and thus yield the complete solution path.

Even if a state, s_c, has been generated earlier, the A* algorithm cannot immediately discard it, since the new version may represent a lower-cost path to s_c. As a simple example, consider the search-tree segment of Figure 5. As described in the caption, state C_1 was first generated as a child of parent state P_1 and later as a child of P_2. Since the latter parent has a lower g value, C_1 can inherit this g improvement and pass it along to its children, who can also recursively pass it to the bottom of the tree.

It is important to note that A* allows the search nodes to connect up into a graph, not simply a tree. Hence, some nodes may have more than one parent, and some of these parents may have higher g values than the

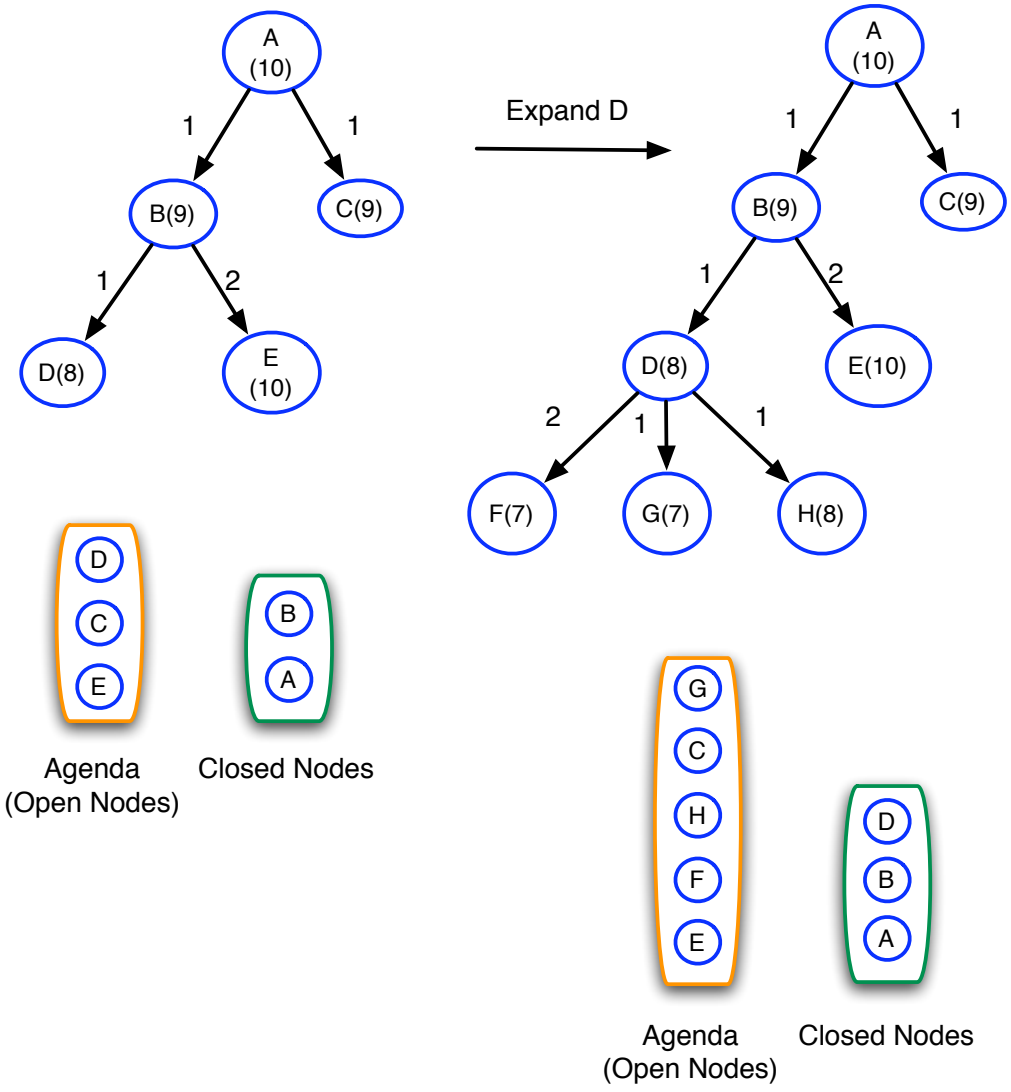

Figure 4: A sample scenario involving a partially-generated search tree along with the lists of open and closed nodes. The open nodes (on the agenda) are sorted by increasing f values, which are the sum of the g values (numbers on the arcs) and h values (numbers in the nodes of the tree). For example, in the agenda on the left, $f(D) = 2 + 8 = 10$, while $f(E) = 3 + 10 = 13$. After expansion of D (right), it is moved to the closed list, while its children's h (and f) values are computed before adding them to the agenda.

child. At any node P, A* records all of P's children, regardless of whether or not P is their best parent (i.e. parent with the lowest g value), since, if a shorter path to P is later discovered, it could become the best parent to many of its children. Within any child, C, A* records the best parent. If A* is being run to produce all solutions, then it keeps track of all parents - a point that we ignore in this document.

So, when propagating an *improvement* from parent P to child C, the algorithm must check whether or not C has a better parent. If P was previously the best parent, then obviously it will remain so with the g-value improvement, so C can incorporate the g change and propagate it to its children. If C had a best parent P*, where P* $\neq$ P, then either:

1. The improved P is now better than P*, in which case P* $\leftarrow$ P and C incorporates and propagates the g improvement, or
2. P* is still as good or better than P, in which case P* remains unchanged and C ignores the g improvement and does not pass it on.

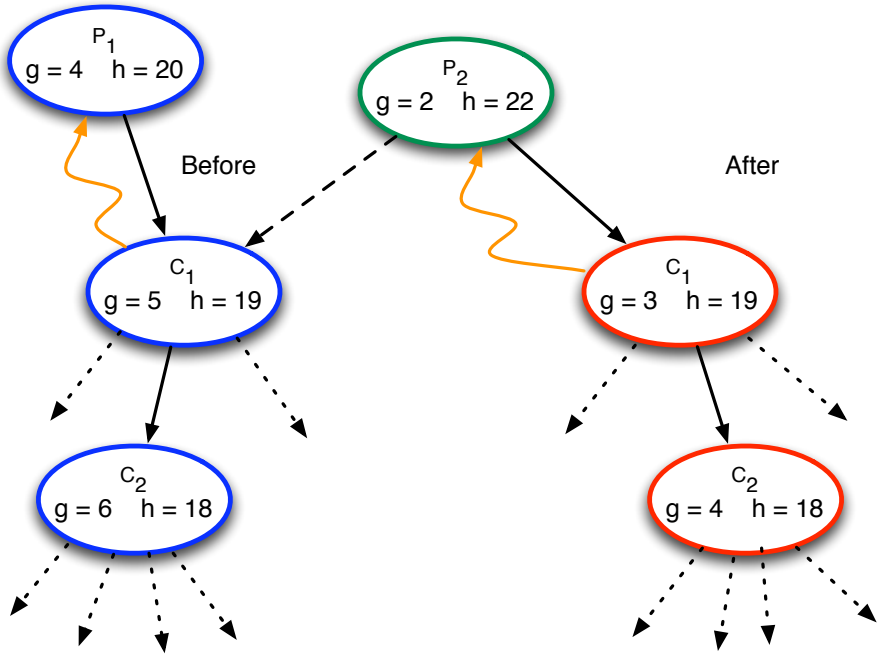

Figure 5: An A* situation in which node C_1 is generated twice, first as the child of P_1 and later as that of P_2. We assume an arc cost of 1 throughout the tree. Since P_2 has a lower g value than P_1, C_1 records P_2 as its *best parent* (wavy line) and updates its own g value from 5 to 3. Furthermore, C_1 propagates the g improvement to all of its descendants, such as C_2.

2.4 Why does this ever matter?

The scenario in Figure 5 often leads to a bit of head-shaking among bright individuals. Why, they ask, would P_1 ever get expanded (or even generated) before P_2. For instance, since P_1 has a higher g value, it makes sense that P_2 would have been generated earlier. However, there is no such guarantee. The true superiority of P_2 does not become evident until AFTER it generates C_1 as a child. At that point, we see

that P_2 provides a more promising route to the goal. We did not know this before, because the heuristic function, h, was not good enough: it gave P_2 a higher h value than P_1, indicating that it thought P_1 was closer to the goal. The heuristic function could well be admissable, but it may still lack enough knowledge to give trustworthy estimates.

Herein lies the confusion. When we think of A* problems, we often think of maze searches and other tasks for which reasonably good (and admissable) heuristics come quickly to mind. In these domains, it seems obvious that if P_2 is one move (with a cost of 1) from C_1, then any semi-intelligent heuristic would give P_2 an h value of $h(C_1) + 1$. Hence, P_2 would have been expanded, to create C_1 long before P_1 was ever generated.

Unfortunately, many domains involve much more complex states for which distances to the goal become extremely difficult to accurately estimate, with or without admissability restrictions. In those domains, a simple move, with a low cost, may have residual affects on the problem state such that the heuristic value suddenly drops precipitously. These *chain reactions* can occur in cases where a problem state consists of a set of logical statements, and the heuristic compares these sets to the logical description of the goal state. Behind the scenes in such a system lies a logical inference mechanism that typically triggers on a new fact (or in the case of logical search, an assumed fact, i.e., assumption). These assumptions often constitute the *moves* of best-first search, and any one move can set off a series of deducations that add (or retract) several logical statements to (from) the state. To foresee all of these deductive consequences can be difficult (or just too computationally expensive) for a heuristic function. So it may greatly over- or under-estimate the distance to a goal state (measured in terms of the number of assumptions needed to get there).

Even if we assume that h is admissable (i.e. never overestimating the distance to the goal), scenarios such as Figure 5 can easily occur if h yields a gross underestimate for C_1 but a slightly less dramatic underestimate for P_2.

So you can ignore situations like that of Figure 5 when building A* to run on a wide variety of problems that admit straightforward heuristic functions, but other, more complex, domains pose so many challenges to heuristics that improved ancestor paths to a state do arise in the course of a run. Two shortcuts are a) ignoring the link between the second parent (e.g. P_2) and child (e.g. C_1) under the assumption that it cannot possibly be any better than the first, and b) never testing child states for equality, in which case there will be multiple occurrences of C_1 **and of the subtree rooted at** C_1 in the OPEN and CLOSED lists. This latter shortcut will still find optimal solutions but can be grossly inefficient, computationally.

On simple problems, you may safely ignore these particular intricasies of the A* algorithm, but for complex problems, their is a risk associated with simple versions of A* (found on some web pages and in some textbooks) that ignore the multiple-parent issue by performing either of the 2 shortcuts listed above.

2.5 A Day in the Life of a Search Node

A search node, n*, in the search graph is generated when a unique new search state, s*, is produced either as the start state or as the immediate successor of an existing state. Once A* confirms that s* is a new state, it generates n*, assigns s* as the search state of n*, and computes f(n*). Next, n* is added to the list of *open* nodes, i.e. the *search-agenda*, at a position determined by f(n*); lower values garner spots closer to the front of the agenda.

As nodes are popped from the agenda and expanded, n* may move closer to the front, although there is no guarantee, since newly-created children may have f values lower than f(n*).

If n* reaches the front of the agenda, it is popped and either:

1. recognized as a solution,
2. recognized as a deadend and then ignored.
3. recognized as an intermediate state and expanded.

When a solution is found, A* normally displays the solution path (from root to solution node) and halts. Alternatively, A* can continue popping and expanding nodes if the goal is to find **all** solutions.

For intermediate nodes, expansion produces child nodes that are immediately linked to the parent, which is then marked as *closed*, and added to the list of closed nodes, called the *closed list*.

At any time during the A* run, s* may be generated again, in which case g(n*), and hence f(n*), may be updated. This can happen when n* is on the agenda or on the closed list. The main difference is that only when n* is on the closed list are there any child nodes for which to recursively propagate g improvements.

2.6 The A* Optimality Claim

An important nuance of the A* algorithm is that although states are generated, evaluated, and placed on the agenda at the same general time in the computation, they are not actually tested for being **solutions** until they are popped from the agenda. This means that solutions can be generated but not *recognized* as solutions until later in the search, if at all.

This algorithmic quirk has strong theoretical implications for A* as an optimal process. To see this, let h*(s) represent the **true** distance from state s to the goal, while h(s) is the distance computed by our A* heuristic function. Then, A* is guaranteed to **recognize** the optimal solution (i.e., pop it off the agenda) before recognizing any other solution, if and only if the following relationship holds:

$$\forall s \in S : h(s) \leq h^{\star}(s) \tag{3}$$

In short, the heuristic function must never overestimate the distance to the goal if A* is to recognize the optimal solution first.

Figure 6 illustrates the basic mechanism by which a non-optimal solution may be generated but never recognized. As shown in the leftmost agenda, V and X may both be open nodes, with f(V) < f(X) - since the distance from V to W may be grossly underestimated by h(V). Thus, V would be expanded first, creating W, a non-optimal solution. However, f(W) = 5, which is larger than f(X). Hence, X will be ahead of W in the agenda, and when X is expanded, Y will also enter the agenda ahead of W. When Y expands and produces Z, this too will enter ahead of W, thus insuring that Z will be popped and recognized as a solution before W.

In summary, the *optimism* of h(s) may easily lead to the generation of a non-optimal solution, W, since the path to it may be replete with gross underestimates of the distance to W. Only when A* actually generates W does the true cost of getting there become known. At that point, W becomes less attractive than any of the nodes along the optimal path (to Z). Thus, W essentially *dies* on the agenda while the optimal path is gradually uncovered.

However, if h(s) were to overestimate the distance-to-goal at any point along the optimal path, then W could bubble to the top of the agenda and be recognized as a solution.

It is now obvious why the claim of A*'s optimality is a bit of an exaggeration. Even when h(s) never overestimates h*(s), the algorithm can generate non-optimal solutions. It just never officially recognizes them as solutions. Although there is some legitimacy in the claim that solution recognition can be a complex process, this weakens considerably in light of the fact that non-optimal solutions are not only generated, but **evaluated** as well. That means that their h values are computed, and since h never overestimates, it must return zero for any solution, optimal or not.

Thus, the only solution-identification process that remains to be performed is the recognition that h(s) is, in fact, a zero, and that this is not an underestimate, i.e., that the state is actually a solution rather than a non-solution that wrongfully received a zero h value. Although devising good heuristics is often difficult, it is normally quite straightforward to insure that they do not give zero values for non solutions. In other words, the states that are normally most amenable to accurate h estimates are those close to the goal, and those farther from the goal are normally easily recognized as non solutions by the heuristic.

In short, given an optimistic heuristic, A* does find optimal solutions, but it cannot strictly avoid generating non-optimal ones. Ironically, this optimism helps create the abundance of non-optimal solutions on the agenda, since nodes such as V (in Figure 6) get expanded precisely because h underestimates distance-to-goal.

2.7 Modifications for Depth- and Breadth-First Search

The A* algorithm is general enough to support depth- and breadth-first search as well. Only a few minor modifications to the handling of the OPEN list are required.

To achieve depth-first search, simply push each newly created (and unique) successor onto the top/front of OPEN. This insures that the most recently-created nodes will be popped first. For breadth-first search, add these successors to the bottom/end of OPEN, thus insuring that an entire level of nodes gets expanded before any nodes in the level below.

Although the heuristic h plays no useful role in depth- nor breadth-first search, the g values can still be helpful. If a node is generated more than once, the g value allows the algorithm to determine which of them is along the more efficient solution path and to adjust the parent pointers accordingly.

No other changes should be necessary.

3 A* and the 25-Checkers Problem

The A* pseudocode above is general-purpose and should be of use in most best-first problem-solving scenarios. However, to apply these general routines to particular problems, a few problem-specific specializations (i.e., subclasses and methods) are necessary. In most cases, this will involve subclasses of a generic *search-state* class, and it will often entail specialization of a high-level class such as *best-first-search*, which manages the overall operation of A* and may require a little tailoring for specific problems. In general, a *search-node* class and its methods for handling a) parent-child node connections, and b) general search-graph creation and maintenance should be sufficient for most A* applications. However, it is often convenient to make a

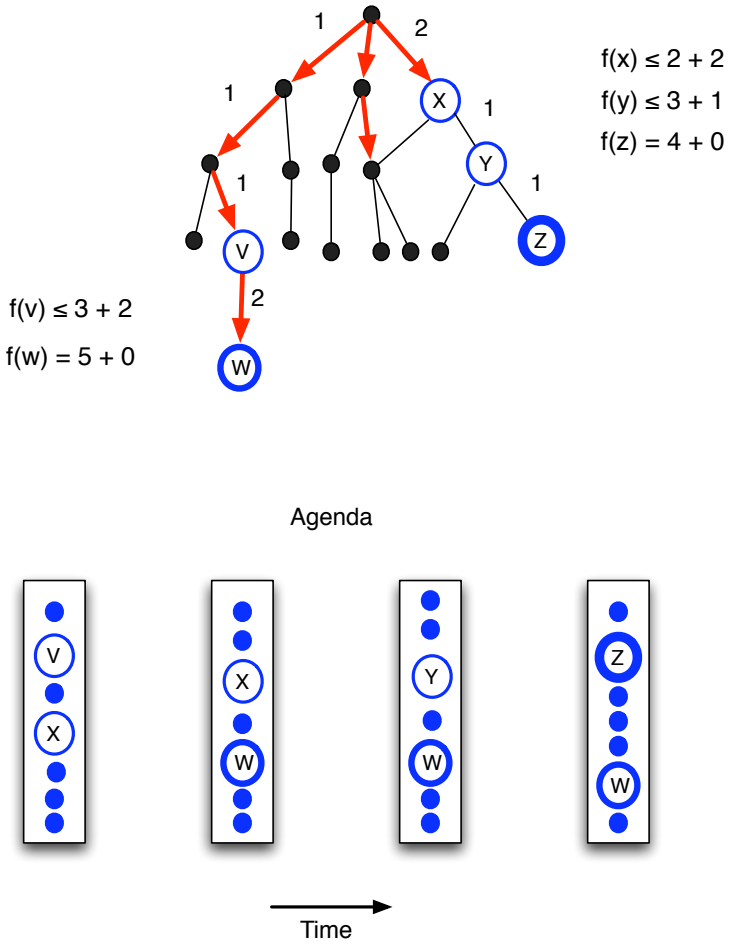

Figure 6: A situation in A* where a non-optimal solution, W, is generated and evaluated but never recognized as a solution. Assume a) arc costs as shown, b) an optimal solution Z, and c) an h(s) that never overestimates h*(s). Then, X and Y will have h values no higher than 2 and 1, respectively. Thus, both will have lower f values than W and be placed ahead of it in the agenda, as shown by the time series of agenda contents. Hence, although W may be generated prior to Z, the latter will be popped from the agenda and recognized as a solution ahead of the former. In the equations, numbers denote values for g(s) and h(s) in f(s) = g(s) + h(s). In the agendas, the small filled circles denote zero or more intervening nodes but have no specific relationship to the small circles in the search tree diagram.

problem-specific subclass anyway, such as *checkers-node*, since properties such as the g and h values will be calculated in problem-specific ways.

To solve the checkers puzzle of Figure 2 using A*, begin by defining subclasses of *search-state* and *best-first-search*, which we will call *checkers-state* and *checkers-bfs*, respectively. Each state should be a complete description of the checkers board at a particular location in search space, for example a 2-dimensional array with cell values corresponding to checkers. The root node of the checkers-bfs object would then house a checkers-state with an array reflecting the board status in Figure 2.

A checkers-bfs object has several important attributes, including *all-swaps*, a list of all legal swap operators that can be applied to a checkers-state. These enable A* to generate all possible successors to a parent state. A *goal-state* slot can be used to hold an array that represents the goal pattern, in this case a checker board with the 25 numbered pieces in sequential order. In practice, this board is not necessary, since the target location for each piece can be calculated directly, but in other cases, the correct checker for a cell may not be easily calculated from the cell indices, thus requiring a direct lookup within the goal state. These 2 slots would need to be filled in during initialization of the checkers-bfs object.

The following methods (whose syntax is simply a function name followed by the argument types in parentheses) are then necessary to tailor A* to the 25-Checkers problem:

1. create-root-node(checkers-bfs) - This creates the initial search state and incorporates it into a node object that becomes the root of the search tree.
2. create-state-identifier(checkers-state) - This converts an entire checker scenario into a large integer to be used for finding the corresponding node in the hash table.
3. generate-successor-states(checkers-bfs, checkers-node) - Given a node in the search tree (or graph), this applies all-swaps to that node's state, with each application producing a different search state, all of which are returned as a collection to the expand-node method.
4. solution?(checkers-bfs, checkers-node) - This compares the checkers-state of the checkers-node to the goal state stored in the checkers-bfs object, returning True only if they match exactly.
5. heuristic-evaluation(checkers-bfs, checkers-node) - This assesses the distance-to-goal of the node's checkers-state. For example, it might sum the Manhattan distances (discussed above) between the current and goal location of each checker. The output of the heuristic evaluation method becomes the h value of the corresponding checkers-node.
6. arc-cost(checkers-bfs,checkers-node,checkers-node) - This calculates the arc cost between two nodes (a parent and child). For the 25-Checkers problem using a heuristic function based on Manhattan distance, some experimentation shows that a constant arc cost of between .05 and 0.2 gives best results.

Most applications of A* require the specialization of most or all of the above methods, but, typically, not many others.

Figure 7 shows the sequence of states from start to goal of the checkers puzzle. In this run, A* expands 6489 nodes; the solution involves 58 of them.

```
? (checker)
 NIL  :#2a((7 24 10 19 3) (12 20 8 22 23) (2 15 25 18 13) (11 21 5 9 16) (17 4 14 1 6))
 ((4 2) (4 3)) : #2a((7 24 10 19 3) (12 20 8 22 23) (2 15 25 18 13) (11 21 5 9 16) (17 4 1 14 6))
 ((4 1) (4 2)) : #2a((7 24 10 19 3) (12 20 8 22 23) (2 15 25 18 13) (11 21 5 9 16) (17 1 4 14 6))
 ((4 0) (4 1)) : #2a((7 24 10 19 3) (12 20 8 22 23) (2 15 25 18 13) (11 21 5 9 16) (1 17 4 14 6))
 ((3 1) (4 1)) : #2a((7 24 10 19 3) (12 20 8 22 23) (2 15 25 18 13) (11 17 5 9 16) (1 21 4 14 6))
 ((2 3) (3 3)) : #2a((7 24 10 19 3) (12 20 8 22 23) (2 15 25 9 13) (11 17 5 18 16) (1 21 4 14 6))
 ((3 2) (3 3)) : #2a((7 24 10 19 3) (12 20 8 22 23) (2 15 25 9 13) (11 17 18 5 16) (1 21 4 14 6))
 ((3 3) (3 4)) : #2a((7 24 10 19 3) (12 20 8 22 23) (2 15 25 9 13) (11 17 18 16 5) (1 21 4 14 6))
 ((1 3) (2 3)) : #2a((7 24 10 19 3) (12 20 8 9 23) (2 15 25 22 13) (11 17 18 16 5) (1 21 4 14 6))
 ((2 2) (2 3)) : #2a((7 24 10 19 3) (12 20 8 9 23) (2 15 22 25 13) (11 17 18 16 5) (1 21 4 14 6))
 ((2 3) (2 4)) : #2a((7 24 10 19 3) (12 20 8 9 23) (2 15 22 13 25) (11 17 18 16 5) (1 21 4 14 6))
 ((2 4) (3 4)) : #2a((7 24 10 19 3) (12 20 8 9 23) (2 15 22 13 5) (11 17 18 16 25) (1 21 4 14 6))
 ((3 4) (4 4)) : #2a((7 24 10 19 3) (12 20 8 9 23) (2 15 22 13 5) (11 17 18 16 6) (1 21 4 14 25))
 ((2 1) (2 2)) : #2a((7 24 10 19 3) (12 20 8 9 23) (2 22 15 13 5) (11 17 18 16 6) (1 21 4 14 25))
 ((2 2) (2 3)) : #2a((7 24 10 19 3) (12 20 8 9 23) (2 22 13 15 5) (11 17 18 16 6) (1 21 4 14 25))
 ((1 4) (2 4)) : #2a((7 24 10 19 3) (12 20 8 9 5) (2 22 13 15 23) (11 17 18 16 6) (1 21 4 14 25))
 ((2 4) (3 4)) : #2a((7 24 10 19 3) (12 20 8 9 5) (2 22 13 15 6) (11 17 18 16 23) (1 21 4 14 25))
 ((2 3) (2 4)) : #2a((7 24 10 19 3) (12 20 8 9 5) (2 22 13 6 15) (11 17 18 16 23) (1 21 4 14 25))
 ((1 0) (2 0)) : #2a((7 24 10 19 3) (2 20 8 9 5) (12 22 13 6 15) (11 17 18 16 23) (1 21 4 14 25))
 ((0 0) (1 0)) : #2a((2 24 10 19 3) (7 20 8 9 5) (12 22 13 6 15) (11 17 18 16 23) (1 21 4 14 25))
 ((3 3) (3 4)) : #2a((2 24 10 19 3) (7 20 8 9 5) (12 22 13 6 15) (11 17 18 23 16) (1 21 4 14 25))
 ((3 3) (4 3)) : #2a((2 24 10 19 3) (7 20 8 9 5) (12 22 13 6 15) (11 17 18 14 16) (1 21 4 23 25))
 ((4 2) (4 3)) : #2a((2 24 10 19 3) (7 20 8 9 5) (12 22 13 6 15) (11 17 18 14 16) (1 21 23 4 25))
 ((2 0) (2 1)) : #2a((2 24 10 19 3) (7 20 8 9 5) (22 12 13 6 15) (11 17 18 14 16) (1 21 23 4 25))
 ((2 0) (3 0)) : #2a((2 24 10 19 3) (7 20 8 9 5) (11 12 13 6 15) (22 17 18 14 16) (1 21 23 4 25))
 ((3 0) (4 0)) : #2a((2 24 10 19 3) (7 20 8 9 5) (11 12 13 6 15) (1 17 18 14 16) (22 21 23 4 25))
 ((4 0) (4 1)) : #2a((2 24 10 19 3) (7 20 8 9 5) (11 12 13 6 15) (1 17 18 14 16) (21 22 23 4 25))
 ((0 3) (0 4)) : #2a((2 24 10 3 19) (7 20 8 9 5) (11 12 13 6 15) (1 17 18 14 16) (21 22 23 4 25))
 ((0 4) (1 4)) : #2a((2 24 10 3 5) (7 20 8 9 19) (11 12 13 6 15) (1 17 18 14 16) (21 22 23 4 25))
 ((0 2) (0 3)) : #2a((2 24 3 10 5) (7 20 8 9 19) (11 12 13 6 15) (1 17 18 14 16) (21 22 23 4 25))
 ((0 3) (1 3)) : #2a((2 24 3 9 5) (7 20 8 10 19) (11 12 13 6 15) (1 17 18 14 16) (21 22 23 4 25))
 ((1 3) (1 4)) : #2a((2 24 3 9 5) (7 20 8 19 10) (11 12 13 6 15) (1 17 18 14 16) (21 22 23 4 25))
 ((1 3) (2 3)) : #2a((2 24 3 9 5) (7 20 8 6 10) (11 12 13 19 15) (1 17 18 14 16) (21 22 23 4 25))
 ((2 3) (3 3)) : #2a((2 24 3 9 5) (7 20 8 6 10) (11 12 13 14 15) (1 17 18 19 16) (21 22 23 4 25))
 ((1 2) (1 3)) : #2a((2 24 3 9 5) (7 20 6 8 10) (11 12 13 14 15) (1 17 18 19 16) (21 22 23 4 25))
 ((1 1) (1 2)) : #2a((2 24 3 9 5) (7 6 20 8 10) (11 12 13 14 15) (1 17 18 19 16) (21 22 23 4 25))
 ((1 2) (1 3)) : #2a((2 24 3 9 5) (7 6 8 20 10) (11 12 13 14 15) (1 17 18 19 16) (21 22 23 4 25))
 ((1 0) (1 1)) : #2a((2 24 3 9 5) (6 7 8 20 10) (11 12 13 14 15) (1 17 18 19 16) (21 22 23 4 25))
 ((1 3) (2 3)) : #2a((2 24 3 9 5) (6 7 8 14 10) (11 12 13 20 15) (1 17 18 19 16) (21 22 23 4 25))
 ((2 3) (3 3)) : #2a((2 24 3 9 5) (6 7 8 14 10) (11 12 13 19 15) (1 17 18 20 16) (21 22 23 4 25))
 ((3 3) (3 4)) : #2a((2 24 3 9 5) (6 7 8 14 10) (11 12 13 19 15) (1 17 18 16 20) (21 22 23 4 25))
 ((3 3) (4 3)) : #2a((2 24 3 9 5) (6 7 8 14 10) (11 12 13 19 15) (1 17 18 4 20) (21 22 23 16 25))
 ((2 3) (3 3)) : #2a((2 24 3 9 5) (6 7 8 14 10) (11 12 13 4 15) (1 17 18 19 20) (21 22 23 16 25))
 ((1 3) (2 3)) : #2a((2 24 3 9 5) (6 7 8 4 10) (11 12 13 14 15) (1 17 18 19 20) (21 22 23 16 25))
 ((0 3) (1 3)) : #2a((2 24 3 4 5) (6 7 8 9 10) (11 12 13 14 15) (1 17 18 19 20) (21 22 23 16 25))
 ((2 0) (3 0)) : #2a((2 24 3 4 5) (6 7 8 9 10) (1 12 13 14 15) (11 17 18 19 20) (21 22 23 16 25))
 ((1 0) (2 0)) : #2a((2 24 3 4 5) (1 7 8 9 10) (6 12 13 14 15) (11 17 18 19 20) (21 22 23 16 25))
 ((0 0) (0 1)) : #2a((24 2 3 4 5) (1 7 8 9 10) (6 12 13 14 15) (11 17 18 19 20) (21 22 23 16 25))
 ((0 0) (1 0)) : #2a((1 2 3 4 5) (24 7 8 9 10) (6 12 13 14 15) (11 17 18 19 20) (21 22 23 16 25))
 ((1 0) (2 0)) : #2a((1 2 3 4 5) (6 7 8 9 10) (24 12 13 14 15) (11 17 18 19 20) (21 22 23 16 25))
 ((2 0) (3 0)) : #2a((1 2 3 4 5) (6 7 8 9 10) (11 12 13 14 15) (24 17 18 19 20) (21 22 23 16 25))
 ((4 2) (4 3)) : #2a((1 2 3 4 5) (6 7 8 9 10) (11 12 13 14 15) (24 17 18 19 20) (21 22 16 23 25))
 ((3 2) (4 2)) : #2a((1 2 3 4 5) (6 7 8 9 10) (11 12 13 14 15) (24 17 16 19 20) (21 22 18 23 25))
 ((3 1) (3 2)) : #2a((1 2 3 4 5) (6 7 8 9 10) (11 12 13 14 15) (24 16 17 19 20) (21 22 18 23 25))
 ((3 0) (3 1)) : #2a((1 2 3 4 5) (6 7 8 9 10) (11 12 13 14 15) (16 24 17 19 20) (21 22 18 23 25))
 ((3 1) (3 2)) : #2a((1 2 3 4 5) (6 7 8 9 10) (11 12 13 14 15) (16 17 24 19 20) (21 22 18 23 25))
 ((3 2) (4 2)) : #2a((1 2 3 4 5) (6 7 8 9 10) (11 12 13 14 15) (16 17 18 19 20) (21 22 24 23 25))
 ((4 2) (4 3)) : #2a((1 2 3 4 5) (6 7 8 9 10) (11 12 13 14 15) (16 17 18 19 20) (21 22 23 24 25))
 6489 nodes expanded
 Depth of solution node: 57
```

Figure 7: A run of the A* algorithm on the 25-Checkers Problem, using half the Manhattan distance as a heuristic and an arc cost of 0.2. Each line begins with two pairs denoting the two pieces that were swapped; the rest of the line is a linearized version of the 2d array, with each row in parentheses.