one, depending upon whether the two characters are the same or different.

Further, the sequence alignment problem can easily be extended to multiple sequences. An *n*-sequence problem will form an *n*-dimensional grid.

5.9 Pruning the *CLOSED* List

One of the functions of the *CLOSED* list is to prevent nodes from being expanded again and again. However, for a node on *CLOSED* to be expanded again, it will have to be generated as a child of some node on *OPEN* that is being expanded. In the *DFS* search algorithms discussed earlier (Figure 2.18), a function *removeSeen* prevented old nodes from being added to *OPEN* again. However, one can also prevent the search from regenerating the *CLOSED* nodes again by observing that for search to "leak back" into *CLOSED* it will have to go through nodes that are children of nodes on *OPEN*. For any node that is being expanded, it suffices to exclude successors that are "behind" the node. An important condition that must be satisfied is the consistency condition. As shown in Section 5.6.1, the consistency condition implies that all nodes in *CLOSED* have the best path to them discovered already. If that is the case then in the *A** algorithm there is no need to generate nodes already in the *CLOSED* again.

5.9.1 Divide-and-Conquer Frontier Search

One way of doing this is to modify the moves that can be made from the nodes on *OPEN* to keep a "tabu" list of disallowed successors for each node that is added to *OPEN*. The move generator is modified such that every time a node *X* (on *OPEN*) is generated as a successor of some node *Y*, *Y* is excluded from becoming a successor of *X*. If the node *X* is generated later as a successor of some other node *Z* then *Z* is also excluded from its list of successors of *X*. That is, every time a node is expanded, it is put on a tabu[3] list of all *its* successors. And when a node is expanded, only the non-tabu successors are generated. As a consequence, every arc in the search graph is traversed only once, and only in one direction. In this way, search can be constrained to only move "forward". An algorithm that uses this approach is called *Divide-and-Conquer Frontier Search* (*DCFS*) (Korf and Zhang, 2000). Along with every node on *OPEN*, the algorithm *DCFS* keeps a list of disallowed moves. The list *CLOSED*, therefore, is no longer needed to prevent the search from leaking back.

The second task of reconstructing the path when the goal is found still remains. *DCFS* addresses this problem by storing a relay node around the halfway mark in the search space for every node on *OPEN*. The

halfway point could be approximately determined when the *g-value* is close to the *h-value* for a node. Every node on *OPEN* that is beyond the halfway mark keeps a pointer to *its* relay node *Relay*, as shown in the Figure 5.24. Note that different nodes on *OPEN* may have different relay nodes.

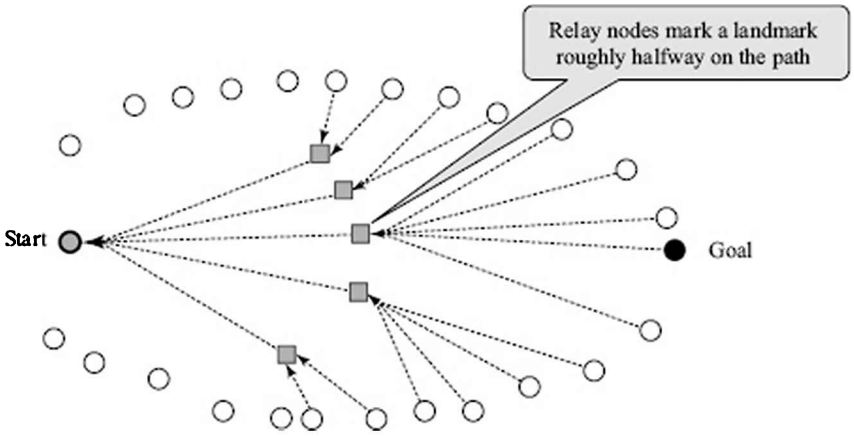

FIGURE 5.24 *Divide-and-Conquer* uses relay nodes to remember landmarks.

When the *DCFS* algorithm picks the goal node, it knows the cost of the solution, but not the path. It has only a pointer to the relay node (*Relay*) on its path, roughly on the halfway mark. To determine the path, *DCFS* is recursively called twice; once to find the path from *Start* to *Relay*, and next to find the path from *Relay* to *Goal*. It has divided the search problem into two parts. This process of recursive calls continues, till the entire path is reconstructed. The algorithm thus saves on space at the expense of running time. If *T(d)* is the time required to search a path of length *d* then *DCFS* has a time complexity given by,

> $T(d) + 2 * T(d/2) + 4 * T(d/4) + \ldots + k * T(d/k)$, where in the last term, $d/k = 1$.

The reader can verify that the above sums up to $T(d) * lg(T(d))$. In the specific case where $T(d)$ is b^d with branching factor b, this becomes $O(b^d * d)$. That is, if one were to do divide-and-conquer reconstruction when search is exponential, one has to do an equivalent of d searches instead of one.

The basic idea of divide and conquer solution reconstruction was adopted from similar techniques used in dynamic programming methods developed earlier for sequence comparison (Hirschberg, 1975; Myers and Miller, 1988) before memory constraints led researchers to look at *A** and its variants.

5.9.2 Sparse-Memory Graph Search

A variation for pruning of the *CLOSED* list takes a different approach. The *Sparse-Memory Graph Search (SMGS)* identifies the *boundary* of the *CLOSED* list, as shown in Figure 5.25 (Zhou and Hansen, 2003). The boundary can be defined as those nodes on *CLOSED* that have at least one neighbour (successor) still on *OPEN*. This can be done by keeping a counter with every node when it is expanded to keep track of the number of children it has on *OPEN*. The counter is decremented each time its children are expanded (the node will appear as a child). As long as the counter is greater than zero, the node is on the boundary. When it becomes zero, it goes into the *kernel*. The nodes of *CLOSED* that are not on the boundary are in the kernel. One can observe that the nodes in the kernel can only be reached via the nodes on the boundary. It would thus be enough to check for new successors to be on the boundary, to prevent the search from leaking back. The nodes in the kernel can be pruned away.

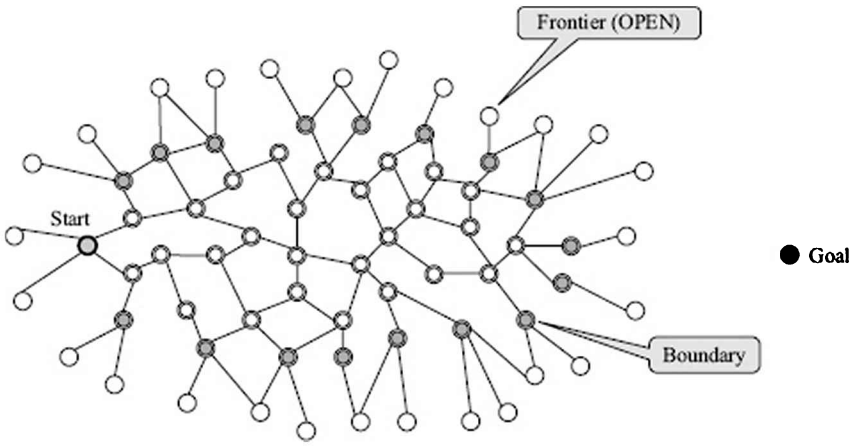

FIGURE 5.25 Boundary nodes in the *CLOSED* list are enough to prevent search from "leaking" back.

The *SMGS* also keeps relay nodes for reconstruction of the path like the *DCFS*. The difference is that it calls a module *PruneClosed* to prune the *CLOSED* list only when it senses that it is running out of memory.

The algorithm identifies three kinds of nodes. One, the *kernel* nodes that have been inspected and all their successors have been inspected. And two, the *boundary* nodes that have been inspected but have some successors on *OPEN*. Finally, the nodes in *OPEN* are the ones that have been generated but have not been inspected. Together, the *kernel* and the *boundary* would form the *CLOSED* set. Initially, the *Start* node is marked as a *relay* node.

The algorithm begins by keeping all three kinds of nodes, and

proceeds to pick nodes from *OPEN* and inspect them. Then at any time if it senses that it is running out of memory, it does the following, by calling a *PruneClosed* function. First, for every node on *OPEN*, it marks the corresponding nodes on the *boundary* as *relay* nodes. Then for each node on the *boundary*, it traces the back pointers to the latest *relay* node, and sets an ancestor pointer to that *relay* node. Then it deletes all *kernel* nodes that are not *relay* nodes. Having finished the pruning, it continues to pick nodes from *OPEN* and inspect them. The first time this is done, the ancestor pointer points to Start. The process is illustrated in Figures 5.26 and 5.27.

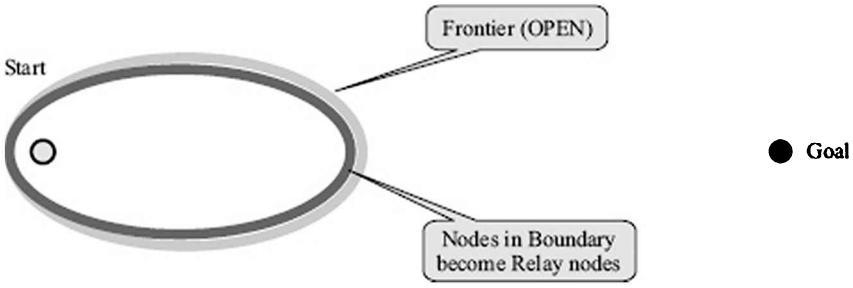

FIGURE 5.26 SMGS converts boundary nodes into Relay when a prune module is called.

The SMGS algorithm may create more than one relay layer if it is solving a large problem. It would have to recursively solve each segment between two consecutive relay nodes; otherwise it continues like the *A** search algorithm. Therefore, while on the one hand it does pruning only when necessary, thus saving on reconstruction cost; on the other hand, it can create many relay layers, thus being able to tackle larger problems.

When the search terminates, there may be several relay layers in the memory. In Figure 5.27 below, we illustrate this with two relay layers. Thus, at the point when search picks the goal node, it also has a Sparse Solution Path to the Start node via the relay nodes. Like the *DCFS* algorithm, *SMGS* recursively calls itself with each segment in the Sparse Solution Path to find the Dense Solution Path, the solution required. While *DCFS* divides the problem into two parts, *SMGS* may divide it in many parts, depending upon how many times the module *PruneClosed* is called. In Figure 5.27, the problem has been divided into three segments, one of which has not yet been pruned.

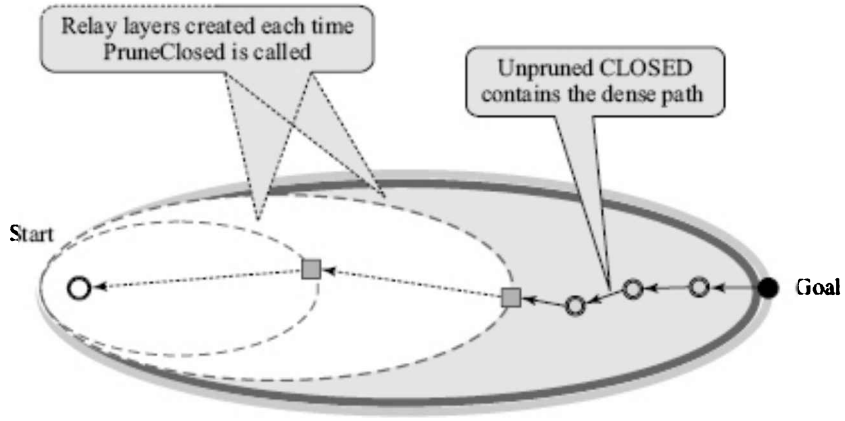

FIGURE 5.27 On termination, *SMGS* has a Sparse Solution Path to the goal.

Of the two search algorithms seen above, *DCFS* prunes *CLOSED* as it goes along. The only nodes it keeps are one relay node for each node on *OPEN*. It does so by modifying the nodes on *OPEN* to keep only forward-going successors. *SMGS* does not tamper with *OPEN*. Instead, it keeps a boundary layer amongst the nodes in *CLOSED* to prevent search from turning backwards. When a call to *PruneClosed* is made, this boundary becomes a relay layer. It also introduces the tactic of pruning only when memory is sensed to be running out, and keeps the option of pruning more than once and creating many relay nodes.

5.10 Pruning the *OPEN* List

In the previous section, we looked at a method to prune the *CLOSED* list. We also observed that it was useful for problems where the problem space only grew polynomially with depth, in which case the list *OPEN* is generally much smaller, often only growing linearly. However, in general, when problem space grows exponentially, it is the *OPEN* list that accounts for most of the memory. In this section, we look at ways to prune the *OPEN* list.

5.10.1 Breadth First Heuristic Search

In the chapter on State Space Search (Chapter 2), we had observed that *Breadth First Search* suffered from an exponentially growing memory requirement. The main reason for that was that the search was uninformed. If we can somehow generate an upper bound U on the solution cost then we could prune away nodes whose f-values are higher than U. This is because f-values are known to be lower bound estimates of solutions containing that node. The upper bound estimate could itself be obtained by using an inexpensive method like *Beam Search*, using

only the heuristic function *h(n)*. The resulting search algorithm called *Breadth First Heuristic Search* (*BFHS*) (Zhou and Hansen, 2004) has been shown to use less memory than *A** search. It explores nodes in a breadth first manner, but prunes nodes that have the estimated cost *f(n)* larger than the upper bound *U*. The following figure suggests why the algorithm keeps a smaller *OPEN* list, and it can be seen that the better the heuristic function, the tighter will be the upper bound.

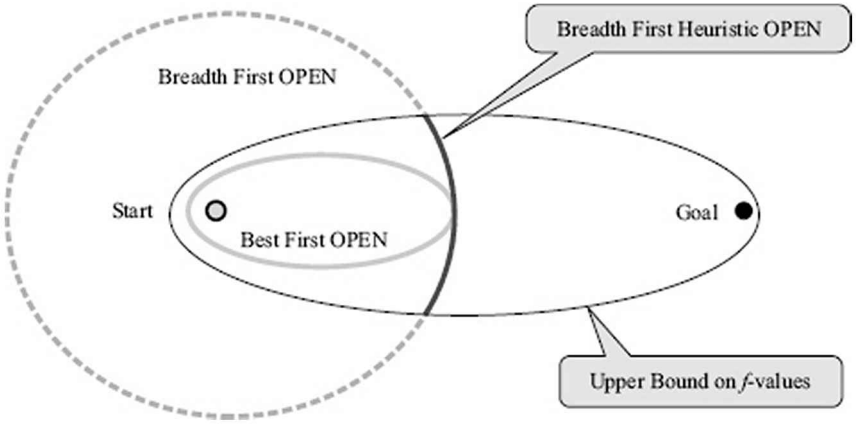

FIGURE 5.28 Breadth First Heuristic Search prunes the OPEN using *f*-values.

Observe that the pruning of nodes from the *Breadth First* frontier is admissible because the pruned nodes cannot be part of the optimal solution. Thus, *BFHS* is a variation that is complete and admissible. It requires lower memory for *OPEN* than *A**, but may expand more nodes than *A** does. The memory required by the *OPEN* of *BFHS* peaks somewhere around the halfway mark. After that point, the number of nodes that have *f*-values within the upper bound starts decreasing. This happens because as one comes closer to the goal, the contribution of the heuristic function *h(n)* to the *f*-values becomes smaller, and the *f*-values become more and more accurate.

5.10.2 Divide and Conquer Beam Search

Beam Search (using *f*-values) can be seen as further pruning the *OPEN* list as shown in Figure 5.29 below. Given that the consistency condition entails that *f*-values increase when one proceeds towards the goal, one has to relax the criterion of moving to only better nodes, and instead move to the best *w* nodes at each level. Since it uses a constant beam width *w*, the memory requirements of *Beam Search* will grow only linearly with depth.

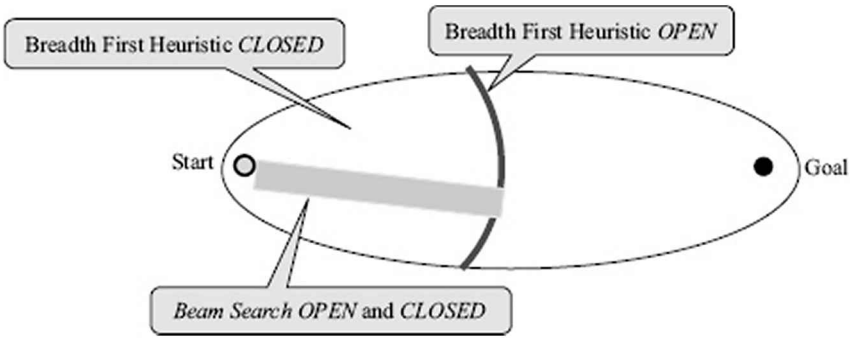

FIGURE 5.29 *Beam Search* further prunes the *OPEN* to a constant width *w*.

The memory required by *Beam Search* is proportional to *dw*, where *d* is the depth and *w* the beam width. However, *Beam Search* is inadmissible, not guaranteeing an optimal solution. In fact, it is not even guaranteed to find any solution. Later, we will address this issue and look at a way to make it complete.

Meanwhile, both *BFHS* and *Beam Search* keep the full *CLOSED* list. If we apply the techniques of pruning the *CLOSED* list from the last section to these two search methods, we get *Divide-and-Conquer BFHS* (*DCBFHS*) and *Divide-and-Conquer Beam Search* (*DCBS*). Beam search expands nodes in a manner similar to breadth first search, except for sorting and pruning each layer to a fixed number of nodes before proceeding to the next. Both may have a partially expanded layer containing both nodes on *OPEN* and *CLOSED*. The next layer will contain a partially formed *OPEN*, while the preceding layer will have only *CLOSED* nodes. The current and the preceding layer will contain the boundary of the search, and are enough to prevent the search from leaking back. In addition, the divide and conquer strategy requires a *RELAY* layer around the halfway point, to which nodes on *OPEN* will hold a pointer to for path reconstruction. Before the search has crossed the halfway mark, the pointer will point to the *Start* node. These four layers are enough for search to progress without regenerating nodes in *CLOSED*. Figure 5.30 illustrates the four layers.

Observe that *Divide-and-Conquer Beam Search* keeps a maximum of *w* elements in each of the four layers. Thus, its memory requirement is 4*w*, which is a constant amount! That means that using *DCBS*, one can search up to any depth using a constant amount of memory. As a corollary, it allows us to fix the beam width *w* as high as resources will allow. The only problem is that it is incomplete.

Next, we look at an approach that allows *Beam Search* to backtrack and try other paths systematically, giving us a complete and admissible search algorithm.

5.10.3 Beam Stack Search

Beam Stack Search (BSS) is essentially beam search with backtracking. One of the reasons for the poor time performance of *IDA** and *RBFS* is that they pursued only one path at a time. In some sense, they underutilised the space available to the algorithm. *Beam Search* allows one to pursue multiple candidates simultaneously, but like *Hill Climbing*, cannot switch paths midcourse, and is therefore incomplete.

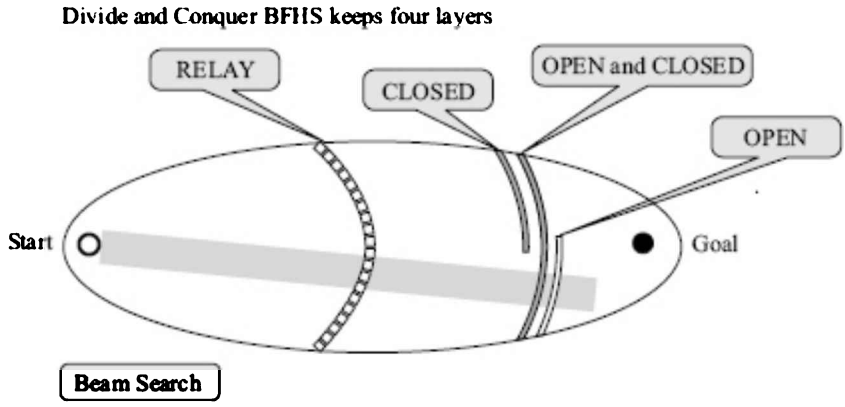

FIGURE 5.30 Divide and Conquer *Beam Search* keeps four layers of constant width.

Beam Stack Search (Zhou and Hansen, 2005) explores the search space *systematically* with a beam of width *w*. Like the *Beam Search*, it too may prune nodes inadmissibly, but it retains the option to backtrack to explore those nodes later. It does this by sorting the nodes at each level on their *f*-values, and keeping track of the minimum and maximum *f*-values of nodes admitted in the beam, at each stage of the search. It does this by keeping a separate stack, called the *Beam Stack*, in which it stores the f_{min} and f_{max} values at each level as a pair [f_{min}, f_{max}) as shown in Figure 5.31 below. Since the algorithm involves sorting of nodes at each level, it is easier to visualise with the search tree it generates.

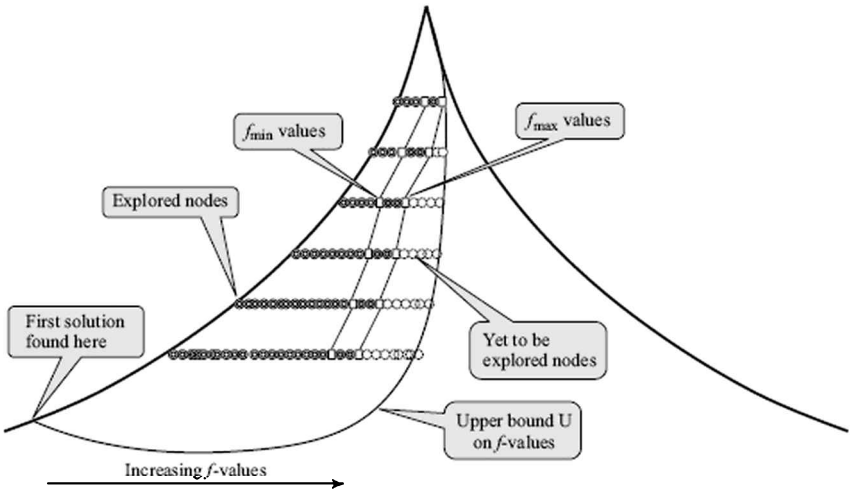

FIGURE 5.31 Beam Stack Search orders nodes on f-values. It slides the f_{min}, f_{max} window on backtracking, but does not go beyond the upper bound U.

The beam stack maintains the f_{min} and f_{max} values at each level. This corresponds to the window in that layer that is exposed to the beam search. The f_{max} value corresponds to the lowest *f*-value node that was pushed out of the beam. When the algorithm backtracks, it slides the corresponding window by setting the new f_{min} to the old f_{max}. It sets the new f_{max} value to the upper bound *U*, updating it to a lower value, only if nodes are generated that cannot be accommodated in the beam width. In this way, it can systematically explore the entire space that *A** would have explored.

Initially, it behaves like *Beam Search*. When it finds a solution, it sets the upper bound value, if a cheaper solution is found. This updating happens every time *BSS* finds a better solution. Once it has found a solution, the solution can be called for at any time. Thus, *BSS* becomes an *anytime algorithm*, quickly finding a good solution which can be returned on demand, but continuing to explore the rest of the space looking for better solutions. The best solution so far can be returned on demand. Moreover, since it uses a heuristic function and explores the lowest *f*-value nodes first, it is quite likely to find the best solution early.

Beam Stack Search is a complete and admissible version of *Beam Search*. Like *Beam Search* (with the *CLOSED* list), the space required by the algorithm is linear with depth.

Finally, the divide and conquer strategy can be applied to *Beam Stack Search* as well, to give a complete and admissible algorithm that requires almost constant space.

5.11 Divide and Conquer Beam Stack Search