11. Memory Limitations in Artificial Intelligence

Stefan Edelkamp

11.1 Introduction

Artificial Intelligence (AI) deals with structuring large amounts of data. As a very first example of an *expert system* [424], take the oldest known scientific treatise surviving from the ancient world, the surgical papyrus [146] of about 3000 BC. It discusses cases of injured men for whom a surgeon had no hope of saving and lay many years unnoticed until it was rediscovered and published for the New York Historical Society. The papyrus summarizes surgical observations of head wounds disclosing an inductive method for inference [281], with observations that were stated with title, examination, diagnosis, treatment, prognosis and glosses much in the sense that *if a patient has this symptom, then he has this injury with this prognosis if this treatment is applied.*

About half a century ago, pioneering computer scientists report the emergence of intelligence with machines that think, learn and create [696]. The prospects were driven by early successes in exploration. Samuel [651] wrote a checkers-playing program that was able to beat him, whereas Newell and Simon [580] successfully ran the *general problem solver* (GPS) that reduced the difference between the predicted and the desired outcome on different state-space problems. GPS represents problems as the task of transforming one symbolic expression into another, with a decomposition that fitted well with the structure of several other problem solving programs. Due to small available memories and slow CPUs, these and some other promising initial AI programs were limited in their problem solving abilities and failed to scale in later years.

There are two basic problems to overcome [539]: the *frame problem* — characterized as *the smoking pistol behind a lot of the attacks on AI* [249] — refers to all that is going on around the central actors, while the *qualification problem* refers to the host of qualifiers to stop an expected rule from being followed exactly. While [454] identifies several arguments of why intelligence in a computer is not a true ontological one, the most important reason for many drawbacks in AI are existing limits in computational resources, especially in memory, which is often too small to keep all information for a suitable inference accessible.

Bounded resources lead to a performance-oriented interpretation of the term intelligence: different to the *Turing-test* [734], programs have to show

U. Meyer et al. (Eds.): Algorithms for Memory Hierarchies, LNCS 2625, pp. 233-250, 2003.

human-adequate or human-superior abilities in a competitive resource-constrained environment on a selected class of benchmarks. As a consequence even the same program can be judged to be more intelligent, when ran on better hardware or when given more time to execute. This competitive view settles; international competitions in data mining (e.g. KDD-Cup), game playing (e.g. Olympiads), robotics (e.g. Robo-Cup), theorem proving (e.g. CADE ATP), and action planning (e.g. IPC) call for highly performant systems on current machines with space and time limitations.

11.2 Hierarchical Memory

Restricted *main memory* calls for the use of *secondary memory*, where objects are either scheduled by the underlying operating system or explicitly maintained by the application program.

Hierarchical memory problems (cf. Chapter 1) have been confronted to AI for many years. As an example, take the *garbage collector problem*. Minsky [551] proposes the first copying garbage collector for LISP; an algorithm using serial secondary memory. The live data is copied out to a file on disk, and then read back in, in a contiguous area of the heap space; [122] extends [551] to parallelize Prolog based on Warren's abstract machine, and modern copy collectors in *C++* [276] also refer to [551]. Moreover, garbage collection has a bad reputation for thrashing caches [439].

Access time graduates on current memory structures: processor register are better available than pre-fetched data, first-level and second level caches are more performant than main memory, which in turn is faster than external data on hard disks optical hardware devices and magnetic tapes. Last but not least, there is the access of data via local area networks and the Internet connections. The faster the access to the memorized data the better the inference.

Access to the next lower level in the memory hierarchy is organized in pages or blocks. Since the theoretical models of hierarchical memory differ e.g. by the amount of disks to be concurrently accessible, algorithms are often ranked according to sorting complexity $\mathcal{O}(\text{sort}(N))$, i.e., the number of block accesses (I/Os) necessary to sort N numbers, and according to scanning complexity $\mathcal{O}(\text{scan}(N))$, i.e., the number of I/Os to read N numbers. The usual assumption is that N is much larger than B, the block size. Scanning complexity equals $\mathcal{O}(N/B)$ in a single disk model. The first libraries for improved secondary memory maintainance are LEDA-SM [226] and TPIE[1]. On the other end, recent developments of hardware significantly deviate from traditional von-Neumann architecture, e.g., the next generation of Intel processors have three processor cache levels. *Cache anomalies* are well known;

[1] `http://www.cs.duke.edu/TPIE`

e.g. recursive programs like Quicksort often perform unexpectedly well when compared to the state-of-the art.

Since the field of improved cache performance in AI is too young and moving too quickly for a comprehensive survey, in this paper we stick to knowledge exploration, in which memory restriction leads to a *coverage problem*: if the algorithm fails to encounter a memorized result, it has to (re)-explore large parts of the problem space. Implicit exploration corresponds to explicit graph search in the underlying problem graph. Unfortunately, theoretical results in external graph search are yet too weak to be practical, e.g. $\mathcal{O}(|V|+\text{sort}(|V|+|E|))$ I/Os for *breadth-first search* (BFS) [567], where $|E|$ is the number of edges and $|V|$ is the number of nodes. One additional problem in external *single-source shortest path* (SSSP) computations is the design of performant external priority queues, for which *tournament-trees* [485] serve as the current best (cf. Chapter 3 and Chapter 4).

Most external graph search algorithms include $\mathcal{O}(|V|)$ I/Os for restructuring and reading the graph, an unacceptable bound for implicit search. Fortunately, for sparse graphs efficient I/O algorithms for BFS and SSSP have been developed (cf. Chapter 5). For example, on planar graphs, BFS and SSSP can be performed in $\mathcal{O}(sort(|V|))$ time. For general BFS, the best known result is $\mathcal{O}\left(\sqrt{|V|\cdot \text{scan}(|V|+|E|)}+\text{sort}(|V|+|E|)\right)$ I/Os (cf. Chapter 4).

In contrast, most AI techniques improve internal performance and include refined state-space representations, increased coverage and storage, limited recomputation of results, heuristic search, control rules, and application-dependent page handling, but close connections in the design of internal space saving strategies and external graph search indicate a potential for cross-fertilization.

We concentrate on *single-agent search*, *game playing*, and *action planning*, since in these areas, the success story is most impressive. Single-agent engines optimally solve challenges like Sokoban [441] and Atomix [417], the 24-Puzzle [468], and Rubik's Cube [466]. Nowadays domain-independent action planners [267, 327, 403] find plans for very large and even infinite mixed propositional and numerical, metric and temporal planning problems. Last but not least, game playing programs challenge human supremacy for example in Chess [410], American Checkers [667], Backgammon [720], Hex [49], Computer Amazons [565], and Bridge [333].

11.3 Single-Agent Search

Traditional single-agent search challenges are puzzles. The "fruit-fly" is the NP-complete (n^2-1)-Puzzle popularized by Loyd and Gardner [325]. Milestones for optimal solutions were [672] for $n=3$, [465] for $n=4$, and [470] for $n=5$. Other solitaire games that are considered to be challenging are

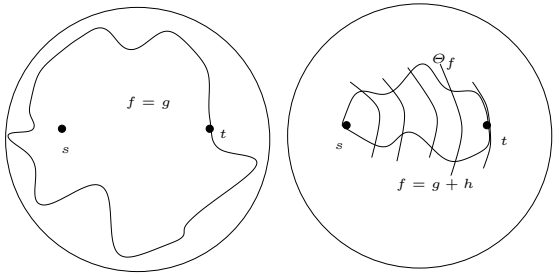

Fig. 11.1. The effect of heuristics in A* and IDA* (right) compared to blind SSSP (left).

the above mentionend PSPACE-hard computer games Sokoban and Atomix. Real-life applications include number partitioning [467], graph partitioning [291], robot-arm motion planning [394], route planning [273], and multiple sequence alignment [777].

Single-agent search problems are either given explicitly in form of a weighted directed graph $G = (V, E, w)$, $w : E \to \mathbb{R}^+$, together with one start node $s \in V$ and (possibly several) goal nodes $T \subseteq V$, or implicitly spanned by a quintuple $(\mathcal{I}, \mathcal{O}, w, \mathit{expand}, \mathit{goal})$ of initial state $\mathcal{I}$, operator set $\mathcal{O}$, weight function $w : \mathcal{O} \to \mathbb{R}^+$, successor generation function *expand*, and goal predicate *goal*. As an additional input, heuristic search algorithms assume an estimate $h : V \to \mathbb{R}^+$, with $h(t) = 0$ for $t \in T$. Since single-agent search can model Turing machine computations, it is undecidable in general [611].

Heuristic search algorithms traverse the re-weighted problem graph. Re-weighting sets the new weight of (u, v) to $w(u, v) - h(u) + h(v)$. The total weight of a path from s to u according to the new weights differs from the old one by $h(s) - h(u)$. Function h is *admissible* if it is a lower bound, which is equivalent to the condition that any path from the current node to the set of goal nodes in the re-weighted graph is of non-negative total weight. Since on every cycle the accumulated weights in the original and re-weighted graph are the same, the transformation cannot lead to negatively weighted cycles. Heuristic h is called *consistent*, if $h(u) \leq h(v) + w(u, v)$, for all $(u, v) \in E$. Consistent heuristics imply positive edge weights.

The *A** algorithm [384] traverses the state space according to a cost function $f(n) = g(n) + h(n)$, where $h(n)$ is the estimated distance from state n to a goal and $g(n)$ is the actual shortest path distance from the initial state. Weighted A* scales between the two extremes; best-first search with $f(n) = h(n)$ and BFS with $f(n) = g(n)$. State spaces are interpreted as implicitly spanned problem graphs, so that A* can be cast as a variant of Dijkstra's SSSP algorithm [252] in the re-weighted graph (cf. Fig. 11.1). In case of negative values for $w(u, v) - h(u) + h(v)$ shorter paths to already expanded nodes may be found later in the exploration process. These nodes

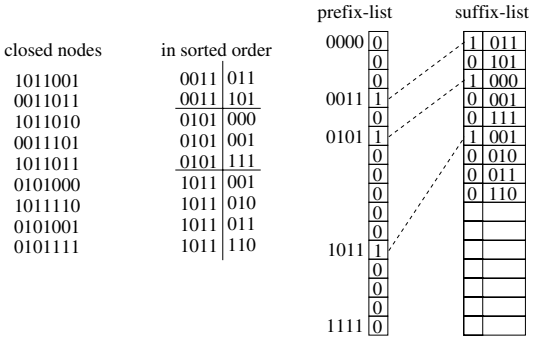

Fig. 11.2. Example for suffix lists with $p = 4$, and $s = 3$.

are *re-opened*; i.e. re-inserted in the set of horizon nodes. Given an admissible heuristic, A* yields an optimal cost path. Despite the reduction of explored space, the main weakness of A* is its high memory consumption, which grows linear with the total number of generated states; the number of expanded nodes $|V'| << |V|$ is still large compared to the main memory capacity of M states.

*Iterative Deepening A** (IDA*) [465] is a variant of A* with a sequence of bounded depth-first search (DFS) iterations. In each iteration IDA* expands all nodes having a total cost not exceeding threshold Θ_f, which is determined as the lowest cost of all generated but not expanded nodes in the previous iteration. The memory requirements in IDA* are linear in the depth of the search tree. On the other hand IDA* searches the tree expansion of the graph, which can be exponentially larger than the graph itself. Even on trees, IDA* may explore $\Omega(|V'|^2)$ nodes expanding one new node in each iteration. Accurate predictions on search tree growth [264] and IDA*'s exploration efforts [469] have been obtained at least for regular search spaces. In favor of IDA*, problem graphs are usually uniformly weighted with an exponentially growing search tree, so that many nodes are expanded in each iteration with the last one dominating the overall search effort.

As computer memories got larger, one approach was to develop better search algorithms *and* to use the available memory resources. The first suggestion was to memorize and update state information also for IDA* in form of a *transposition table* [631]. Increased coverage compared to ordinary hashing has been achieved by *state compression* and by *suffix lists*. State compression minimizes the state description length. For example the internal representation of a state in the 15-Puzzle can easily be reduced to 64 bits, 4 bits for each tile. Compression often reduces the binary encoding length to $\mathcal{O}(\log |V|)$, so that we might assume that for constant c the states u to be stored are assigned to a number $\phi(u)$ in $[1, \ldots, n = |V|^c]$. For the 15-Puzzle the size of the state space is $16!/2$, so that $c = 64/\lceil \log(16!/2) \rceil = 64/44 \approx 1.45$.

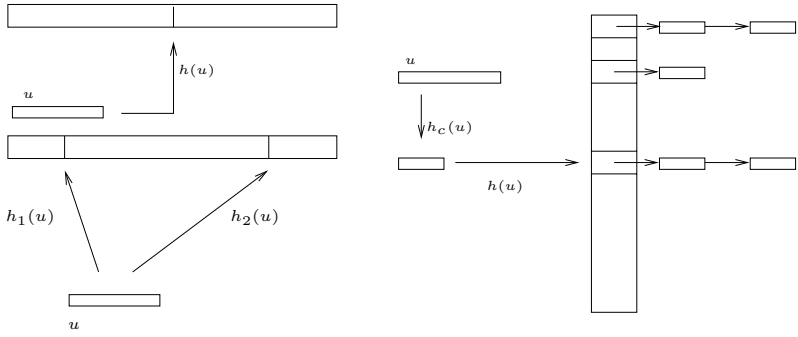

Fig. 11.3. Single bit-state hashing, double bit-state hashing, and hash-compact.

Suffix lists [271] have been designed for external memory usage, but show a good space performance also for internal memorization. Let $bin(\phi(u))$ be the binary representation of an element u with $\phi(u) \leq n$ to be stored. We split $bin(\phi(u))$ in p high order bits and $s = \lceil \log n \rceil - p$ low order bits. Furthermore, $\phi(u)_{s+p-1}, \ldots, \phi(u)_s$ denotes the prefix of $bin(\phi(u))$ and $\phi(u)_{s-1}, \ldots, \phi(u)_0$ stands for the suffix of $bin(u)$. The suffix list consists of a linear array P and of a two-dimensional array L. The basic idea of suffix lists is to store a common prefix of several entries as a single bit in P, whereas the distinctive suffixes form a group within L. P is stored as a bit array. L can hold several groups with each group consisting of a multiple of $s + 1$ bits. The first bit of each $(s + 1)$-bit row in L serves as a *group bit*. The first s bit suffix entry of a group has group bit one, the other elements of the group have group bit zero. We place the elements of a group together in lexicographical order, see Fig. 11.2. The space performance is by far better than ordinary hashing and very close to the information theoretical bound. To improve time performance to amortized $\mathcal{O}(\log |V|)$ for insertions and memberships, the algorithm buffers states and inserts checkpoints for faster prefix-sum computations.

Bit-state hashing [224] and *state compaction* reduce the state vector size to a selection of few bits allowing even larger table sizes. Fig. 11.3 illustrates the mapping of state u via the hash functions h, h_1 and h_2 and compaction function h_c to the according storage structures. This approach of *partial search* necessarily sacrifices completeness, but often yields shortest paths in practice [417]. While hash compact also applies to A*, single and double bit-state hashing are better suited to IDA* search [271], since the priority of a state and its predecessor pointer to track the solution, are mandatory for A*.

In regular search spaces, with a finite set of different operators to be applied, *Finite state machine (FSM) pruning* [715] provides an alternative for duplicate prediction in IDA*. FSM pruning pre-computes a string acceptor for move sequences that are guaranteed to have shorter equivalents; the set of *forbidden words*. For example, twisting two opposite sides of the Rubiks cube in one order, has always an equivalent in twisting them in the opposite order.

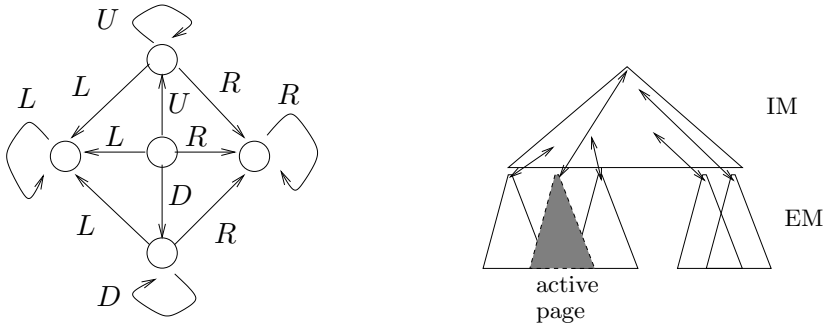

Fig. 11.4. The finite state machine to prune the Grid (left) and the heap-of-heap data structure for localized A* (right). The main and the active heap are in internal memory (IM), while the others reside on external memory (EM).

This set of forbidden words is established by hash conflicts in a learning phase prior to the search and converted to a substring acceptor by the algorithm of Aho and Corasick [20]. Fig 11.4 shows an example to prune the search tree expansion in a regular Grid. The FSM enforces to follow the operators *up* (U), *down* (D), *left* (L), and *right* (R) along the corresponding arrows reducing the exponentially sized search tree expansion with 4^d states, $d > 0$, to the optimum of d^2 states. Suffix-tree automata [262] interleave FSM construction and usage.

In route planning based on spatial partitioning of the map, the heap-of-heap priority-queue data structure of Fig. 11.4 has effectivly been integrated in a localized A* algorithm [273]. The map is sorted according to the two dimensional physical layout and stored in form of small heaps, one per page, and one being active in main memory. To improve locality in the A* derivate, *deleteMin* is substituted by a specialized *deleteSome* operation that prefers node expansions with respect to the current page. The algorithm is shown both to be optimal and to significantly reduce page faults counter-balanced with a slight increase in the number of node expansions. Although locality information is exploited, parts of the heap-of-heap structure may be substituted by provably I/O efficient data structures (cf. Chapter 2 and Chapter 3) like *buffer trees* [54] or *tournament trees* [485].

Most *memory-limited search* algorithms base on A*, and differ in the caching strategies when memory becomes exhausted. MREC [684] switches from A* to IDA* if the memory limit is reached. In contrast, SMA* [645] reassigns the space by dynamically deleting a previously expanded node, propagating up computed f-values to the parents in order to save re-computation as far as possible. However, the effect of node caching is still limited. An adversary may request the nodes just deleted. The best theoretical results on search trees are $O(|V'| + M + |V'|^2/M)$ node expansions in the MEIDA* search algorithm [260]. The algorithm works in two phases: The first phase fills the doubly-ended priority queue D with at most M nodes in IDA* man-

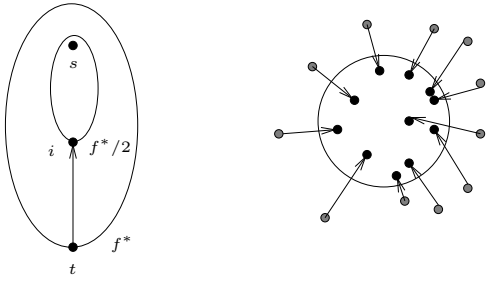

Fig. 11.5. Divide step in undirected frontier search (left) and backward arc look-ahead in directed frontier search (right).

ner. These nodes are expanded and re-inserted into the queue if they are *safe*, i.e., if D is not full and the f-value of the successor node is still smaller than the maximal f-value in D. This is done until D eventually becomes empty. The last expanded node then gives the bound for the next IDA* iteration. Let $E(i)$ be the number of expanded nodes in iteration i and $R(i) = E(i) - E(i-1)$ the number of newly generated nodes in iteration i. If l is the last iteration then the number of expanded nodes in the algorithm is $\sum_{i=1}^{l} i \cdot R(l-i+1)$. Maximizing $\sum_{i=1}^{l} i \cdot R(l-i+1)$ with respect to $R(1)+\ldots+R(l) = E(l) = |V'|$, and $R(i) \geq M$ for fixed $|V'|$ and l yields $R(l) = 0$, $R(1) = |V'| - (l-2)M$ and $R(i) = M$, for $1 < i < l$. Hence, the objective function is maximized at $-Ml^2/2 + (|V'| + 3M/2)l - M$. Maximizing for l yields $l = |V'|/M + 3/2$ and $O(|V'| + M + |V'|^2/M)$ nodes in total.

Frontier search [471] contributes to the observation that the newly generated nodes in any graph search algorithm form a connected horizon to the set of expanded nodes, which is omitted to save memory. The technique refers to Hirschberg's linear space divide-and-conquer algorithm for computing maximal common sequences [400]. In other words, frontier search reduces a $(d+1)$-dimensional search problem into a d-dimensional one. It divides into three phases: in the first phase, a goal t with optimal cost f^* is searched; in the second phase the search is re-invoked with bound $f^*/2$; and by maintaining shortest paths to the resulting fringe the intermediate state i from s to t is detected, in the last phase the algorithm is recursively called for the two subproblems from s to i, and from i to t. Fig. 11.5 depicts the recursion step and indicates the necessity to store virtual nodes in directed graphs to avoid falling back behind the search frontier, where a node v is called *virtual*, if $(v, u) \in E$, and u is already expanded.

Many external exploration algorithms perform variants of frontier search. In the $\mathcal{O}(|V'| + \text{sort}(|V'| + |E'|))$ I/O algorithm of Munagala and Ranade [567] the set of visited lists is reduced to one additional layer. In difference to the internal setting above, this algorithm performs a complete exploration and uses external sorting for duplicate elimination.

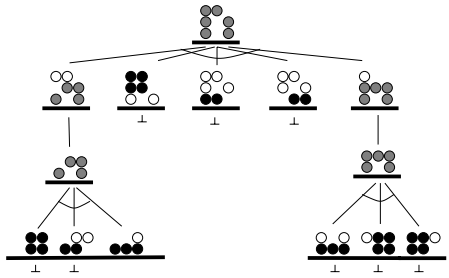

Fig. 11.6. Bootstrapping to build dead-end recognition tables.

Large *dead-end recognition tables* [441] are best built in sub-searches of problem abstractions and avoid non-progressing exploration. Fig. 11.6 gives an example of bootstrapping dead-end patterns by expansion and decomposition in the *Sokoban* problem, a block-sliding game, in which blocks are only to be pushed from an accessible side. Black ball patterns are found by simple recognizers, while gray ball patterns are inferred in bottom-up fashion. Established sub-patterns subsequently prune exploration.

Another approach to speed up the search is to look for more accurate estimates. With a better heuristic function the search will be guided faster towards the goal state and has to deal with less of nodes to be stored. Problem-dependent estimates may have a large overhead to be computed for each encountered state, calling for a more intelligent usage of memory. Perimeter Search [253] is an algorithm that saves a large table in memory which contains all nodes that surround the goal node up to a fixed depth. The estimate is then defined as the distance of the current state and the closest state in the perimeter.

11.4 Action Planning

Domain-independent action planning [29] is one of the central problems in AI, which arises, for instance, when determining the course of action for a robot. Problem domains and instances are usually specified in a general domain description language (PDDL) [312]. Its "fruit-flies" are *Blocks World* and *Logistics*. Planning has effectively been applied for instance in robot navigation [367], elevator scheduling [462], and autonomous spacecraft control [322].

A classical (grounded) *Strips planning problem* [303] is formalized as a quadruple $\mathcal{P} =< \mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G} >$, with $\mathcal{S} \subseteq 2^F$ being the set of states, 2^F being the power set of the set of propositional atoms F, $\mathcal{I} \in \mathcal{S}$, $\mathcal{G} \subseteq \mathcal{S}$, and $\mathcal{O}$ being the set of operators that transform states into states. A state $S \in \mathcal{S}$ is interpreted by the conjunct of its atoms. Operator $o = (P, A, D) \in \mathcal{O}$ is specified with its precondition, add and delete lists, $P, A, D \subseteq F$. If $P \subseteq S$,

the result $S' \in \mathcal{S}$ of an operator $o = (P, A, D)$ applied to state $S \in \mathcal{S}$ is defined as $S' = (S \setminus D) \cup A$. *Mixed propositional and numerical planning* [312] take $\mathcal{S} \subseteq 2^F \times \mathbb{R}^k$, $k > 0$, as the set of states, *temporal planning* includes a special variable *total-time* to fix action execution time, and *metric planning* optimizes an additionally specified objective function. Strips planning is PSPACE-complete [167] and *mixed propositional and numerical planning* is undecidable [391].

Including a non-deterministic choice on actions effects is often used to model uncertainty of the environment. *Strong plans* [204], are plans that guarantee goal achievement despite all non-determinism. *Strong plans* are complete compactly stored state-action tables, that can be best viewed as a controller, that applies certain actions depending on the current state. In contrast, in *conformant planning* [203] a plan is a simple sequence of actions, that is successful for all non-deterministic behaviours. Planning with partial observability interleaves action execution and sensing. In contrast to the successor set generation based on action application, observations correspond to "And" nodes in the search tree [120]. Both conformant and *partial observable planning* can be cast as a deterministic traversal in *belief space*, defined as the power set $2^{\mathcal{S}}$ of the original one planning state space $\mathcal{S}$ [142]. Belief spaces and complete state-action tables are seemingly too large to be explicitly stored in main memory, calling for refined internal representation or fast external storage devices.

In *probabilistic planning* [142], different action outcomes are assigned to a probability distribution and resulting plans/policies correspond to complete state-action tables. Probabilistic planning problems are often modeled as Markov decision process (MDPs) and mostly solved either by policy or value iteration, where the latter invokes successive updates to Bellmann's equation. The complexity of probabilistic planning with partial observability is PP^{NP}-complete [568]. Different caching strategies for solving larger partial observable probabilistic planning problems are studied in [526], with up to substantial CPU time savings for application dependent caching schemes.

Early planning approaches in Strips planning were able to solve only small Strips problems, e.g., to stack five blocks, but planning graphs, SAT-encodings, as well as heuristic search have changed the picture completely. *Graphplan* [132] constructs a layered planning graph containing two types of nodes, action nodes and proposition nodes. In each layer the preconditions of all operators are matched, such that *Graphplan* considers instantiated actions at specific points in time. *Graphplan* generates partially ordered plans to exhibit concurrent actions and alternates between two phases: *graph extension* to increase the search depth and *solution extraction* to terminate the planning process. *Satplan* [452] simulates a BFS according to the binary encoding of planning states, with a standard representation of Boolean formulae as a conjunct of clauses.

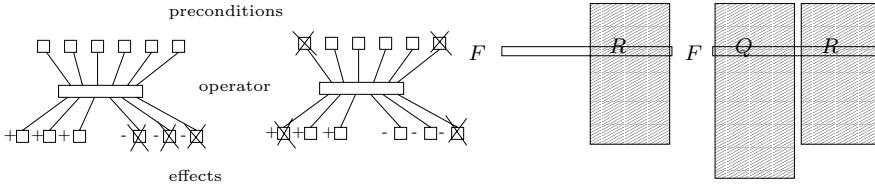

Fig. 11.7. Operator abstractions for the relaxed planning and the pattern database heuristic (left); single and disjoint PDB for subsets R and Q of all atoms F (right).

In current heuristic search planning, relaxed plans [403] and pattern databases (PDB) [266] turn out to be best (cf. Fig. 11.7). The *relaxed planning heuristic* generates approximate plans to the simplified planning problem, in which negative effects have been omitted in all operators, and is computed in a combined forward and backward traversal for each encountered state. The first phase determines a fix point on the set of reachable atoms similar to *Graphplan*, while the second phase greedily extracts the approximate plan length as a heuristic estimate.

Pattern databases (PDB) [232] are constant-time look-up tables generated by completely explored problem abstractions. Subgoals are clustered and for all possible combination of these subgoals in every state of the problem, the relaxed problem is solved without looking on the other subgoals. Fig. 11.7 illustrates PDB construction. The planning state is represented as a set of propositional atoms F, and the operators are projected to one or several disjoint subsets of F. The simplified planning problem is completely explored with any SSSP algorithm, starting from the set of goal states. The abstract states are stored in large hash tables together with their respective goal distance, to serve as heuristic estimates for the overall search. Retrieved values of different databases are either maximized or added. PDBs optimally solved the 15-Puzzle [232] and Rubik's Cube [466]. A space-time trade-off for PDB is analyzed in [405]; PDB size is shown to be inversely correlated to search time. Since search time is proportional to the number of expanded nodes N' and PDB-size is proportional to M, PDBs make very effective use of main memory. Finding good PDB abstractions is not immediate. The general search strategy for optimal-sized PDBs [405] applies to a large set of state-space problems and uses IDA* search tree prediction formula [469] as a guidance. A general bin-packing scheme to generate disjoint PDBs [469] is proposed in [263]. Recall that disjoint PDBs generate very fast optimal solutions to the 24-Puzzle.

Some planners cast planning as *model checking* [334] and apply *binary decision diagrams*(BDDs) [163] to reduce space consumption. BDDs are compact acyclic graph data structures to represent and efficiently manipulate Boolean functions; the nodes are labeled with Boolean variables with two outgoing edges corresponding to the two possible outcomes when evaluating a given assignment, while the 0- or 1-sink finally yield the determined re-

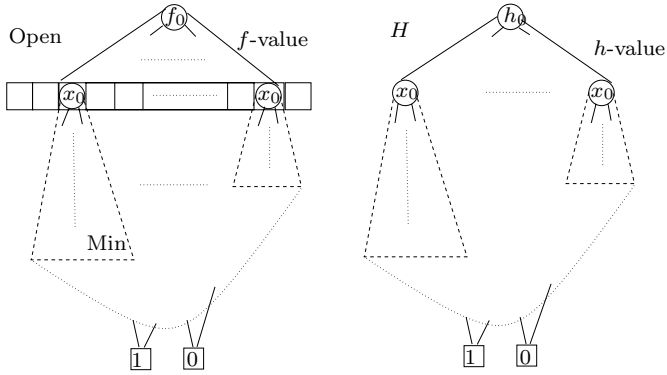

Fig. 11.8. Symbolic heuristic A* search with symbolic priority queue and estimate.

sult. *Symbolic exploration* refers to *Satplan* and realizes a BFS exploration on BDD representations of sets of states, where a state is identified by its characteristic function. Given the represented set of states $S_i(x)$ in iteration i and the represented transition relation T the successor set $S_{i+1}(x)$ is computed as $S_{i+1}(x) = \exists x\ (S_i(x) \wedge\ T(x, x'))[x/x']$. In contrast to conjunctive partitioning in hardware verification [544], for a refined image computation in symbolic planning disjunctive of the transition function is required: if $T(x, x') = \vee_{o \in \mathcal{O}} o(x, x')$ then $S_{i+1}(x) = \exists x\ (S_i(x) \wedge\ \vee_{o \in \mathcal{O}} o(x, x'))[x/x'] = \vee_{o \in \mathcal{O}} (\exists x\ (S_i(x) \wedge\ o(x, x')))[x/x']$.

Symbolic heuristic search maintains the search horizon *Open* and the heuristic estimate H in compact (monolithical or partitioned) form. The algorithm BDDA* [272] steadily extracts, evaluates, expands and re-inserts the set of states *Min* with minimum f-value (cf. Fig. 11.8). For consistent heuristics, the number of iterations in BDDA* can be bounded by the square of the optimal path length.

Although *algebraic decision diagrams* (ADDs), that extend BDDs with floating point labeled sinks, achieve no improvement to BDDs to guide a symbolic exploration in the 15-Puzzle [380], generalization for probabilistic planning results in a remarkable improvement to the state-of-the-art [292]. Pattern databases and symbolic representations with BDDs can be combined to create larger look-up tables and improved estimates for both explicit and symbolic heuristic search [266]. In conformant planning BDDs also apply best, while in this case heuristics have to trade information for exploration gain [119].

Since BDDs are also large graphs, improving memory locality has been studied e.g. in the breadth-first synthesis of BDDs, that constructs a diagram levelwise [412]. There is a trade-off between memory overhead and memory access locality, so that hybrid approaches based on context switches have been explored [772]. An efficiency analysis shows that BDD reduction of a decision diagram G can be achieved in $\mathcal{O}(\text{sort}(|G|))$ I/Os, while Boolean

operator application to combine two BDDs G_1 and G_2 can be performed in $\mathcal{O}(\text{sort}(|G_1| + |G_2|))$ I/Os [53].

Domain specific information can control forward chaining search [78]. The proposed algorithm progresses first order knowledge through operator application to generate an extended state description and may be interpreted as a form of parameterized FSM pruning.

Another space efficient representation for depth-first exploration of the planning space is a *persistent search tree* [78], storing and maintaining the set of instantiations of planning predicates and functions. Recall that persistent data structures only store differences of states, and are often used for text editors or version management systems providing fast and memory-friendly random access to any previously encountered state.

Mixed propositional, temporal and numerical planning aspects call for plan schedules, in which each action is attached to a fixed time-interval. In contrast to ordinary scheduling the duration of an action can be state-dependent. The currently leading approaches[2] are an interleaved plan generator and optimal (PERT) scheduler of the imposed causal structure (MIPS), and a local search engine on planning graphs, optimizing plan quality by deleting and adding actions of generated plans governed by Lagrange multipliers (LPG).

Static analysis of planning domains [311] leads to a general efficient state compression algorithm [268] and is helpful in different planners, especially in BDD engines. *Generic type* analysis of domain classes [515] drives the design of hybrid planners, while different forms of symmetry reduction based on object isomorphisms, effectively shrink exploration space [313]: generic types exploit similarities of different domain structures, and symmetry detection utilizes the parametric description of domain predicates and actions.

Action planning is closely related to *error detection* in software [269] and hardware designs [629], where systems are modeled as state transition graphs of synchronous or asynchronous systems and analyzed by reasoning about properties of states or paths. As in planning, the central problem is overcoming combinatorial explosion; the number of system states is often exponential in the number of state variables. The transfer of technology is rising: *Bounded model checking* [124] exports the *Satplan* idea to error detection, *symbolic model checking* has lead to BDD based planning, while *directed model checking* [269] matches with the success achieved with heuristic search planning.

One approach that has not yet been carried over is *partial order reduction* [509], which compresses the state space by of avoiding concurrent actions, thus reducing the effective branching factor. In difference to FSM pruning, partial ordering sacrifices optimality, detects necessary pruning conditions on the fly, and utilizes the fact that the state space is composed by the cross product of smaller state spaces containing many local operators.

[2] www.dur.ac.uk/d.p.long/competition.html

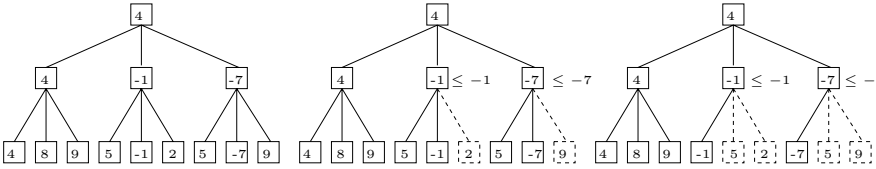

Fig. 11.9. Mini-max game search tree pruned by $\alpha\beta$ and additional move ordering.

11.5 Game Playing

One research area of AI that has ever since dealt with given resource limitations is game playing [666]. Take for example a *two-payer zero-sum game* (with perfect information) given by a set of states $\mathcal{S}$, move-rules to modify states and two players, called Player 0 and Player 1. Since one player is active at a time, the entire state space of the game is $\mathcal{Q} = \mathcal{S} \times \{0, 1\}$. A game has an initial state and some predicate *goal* to determine whether the game has come to an end. We assume that every path from the initial state to a final one is finite. For the set of goal states $\mathcal{G} = \{s \in \mathcal{Q} \mid \mathit{goal}(s)\}$ we define an evaluation function $v : \mathcal{G} \to \{-1, 0, 1\}$, -1 for a lost position, 1 for a winning position, and 0 for a draw. This function is extended to $\hat{v} : \mathcal{Q} \to \{-1, 0, 1\}$ asserting a game theoretical value to each state in the game. More general settings are *multi-player games* and *negotiable games* with incomplete information [628].

DFS dominates game playing and especially computer chess [531], for which [387] provides a concise primer, including mini-max search, $\alpha\beta$ pruning, minimal-window and quiescence search as well as iterative deepening, move ordering, and forward pruning. Since game trees are often too large to be completely generated in time, static evaluation functions assert numbers to root nodes of unexplored subtrees. Fig. 11.9 illustrates a simple mini-max game tree with leaf evaluation, and its reduction by $\alpha\beta$ *pruning* and move ordering. In a game tree of height h with branching factor b the minimal traversed part tree reduces from size $\mathcal{O}(b^h)$ to $\mathcal{O}(\sqrt{b^h})$. Quiescence search extends evaluation beyond exploration depth until a quiescent position is reached, while *forward pruning* refers to different unsound cut-off techniques to break full-width search. *Minimal window search* is another inexact approximation of $\alpha\beta$ with higher cut-off rates.

As in single-agent search, *transposition tables* are memory-intense containers of search information for valuable reuse. The stored move always provides information, but the memorized score is applicable only if the nominal depth does not exceed the value of the cached draft.

Since the early 1950s, from the "fruit-fly"-status, *Chess* has advanced to one of the main successes in AI, resulting in the defeat of the human-world champion in a tournament match. DeepThought [532] utilized IBM's DeepBlue architecture for a massive-parallelized, hardware-oriented $\alpha\beta$ search scheme, evaluating and storing billions of nodes within a second, with a fine-

tuned evaluation function and a large, man-made and computer-validated opening book.

Nine-Men-Morris has been solved with huge *endgame databases* (EDB) [326], in which every state after the initial placement has been asserted to its game-theoretical value. The outcome of a complete search is that the game is a draw. Note that the DeepBlue chess engine is also known to have held complete EDB on the on-chip memories.

Four Connect has been proven to be a win for the first player in optimal play using a knowledge-based approach [31] and mini-max-based *proof number search* (PNS) [32], that introduces the third value *unknown* into the game search tree evaluation. PNS has a working memory requirement linear in the size of the search tree, while $\alpha\beta$ requires only memory linear to the depth of the tree. To reduce memory consumption [32], solved subtrees are removed, or leveled execution is performed. PNS also solved GoMoku, where the search tree is partitioned into a few hundred subtrees, externally stored and combined into a final one [32]. *Proof Set Search* is a recent improvement to PNS, that saves node explorations for a more involved memory handling [564].

Hex is another PSPACE complete board game invented by the Danish mathematician Hein [161]. Since the game can never result in a draw it is easy to prove that the game is won for the first player to move, since otherwise he can adopt the winning strategy of the second player to win the game. The current state-of-the-art program *Hexy* uses a quite unusual approach electrical circuit theory to combine the influence of sub-positions (virtual connections) to larger ones [49].

Go has been addressed by different strategies. One important approach [562] with exponential savings in some endgames uses a divide-and-conquer method based on *combinatorial game theory* [116] in which some board situations are split into a sum of local games of tractable size. *Partial order bounding* [563] propagates relative evaluations in the tree and has also been shown to be effective in Go endgames. It applies to all mini-max searchers, such as $\alpha\beta$ and PNS.

An alternative to $\alpha\beta$ search with minimax evaluation is *conspiracy number search* (CNS) [537]. The basic idea of CNS is to search the game tree in a manner that at least $c > 1$ leaf values have to change in order to change the root one. CNS has been successfully applied to chess [516, 665].

Memory limitation is most apparent in the construction of EDBs [743]. Different to analytical machine learning approaches [673], that construct an explanation why the concept being learned works for positive learning examples — to be stored in operational form for later re-use in similar cases — EDBs do not discover general rules for infallible play, but are primary sources of information for the game-theoretical value of the respective endgame positions. The major compression schemes for positions without pawns use symmetries along the axes of the chess board.

EDB can also be constructed with symbolic, BDD-based exploration [89, 265], but an improving integration of symbolic EDBs in game playing has still to be given. Some combinatorial chess problems like the total number of 33,439,123,484,294 complete Knight's tours have been solved with the compressed representation of BDDs [513].

For EDBs to fit into main memory the general principle is to find efficient encodings. For external usage run-length encoding suits best for output in a final external file [491]. Huffman encodings [215] are further promising candidates. Thereby, modern game playing programs quickly become I/O bound, if they probe external EDBs not only at the root node. In checkers [666] the distributed generation of a very large EDBs has given the edge in favor to the computer. Schaeffer's checker program *Chinook* [667] has perfect EDB information for all checker positions involving eight or fewer pieces on the board, a total of 443,748,401,247 positions generated in large retrograde analysis using a network of workstations and various high-end computers [491]. Commonly accessed portions of the database are pre-loaded into memory and have a greater than 99% hit rate with a 500MB cache [481]. Even with this large cache, the sequential version of the program is I/O bound. A parallel searching version of *Chinook* further increased the I/O rate such that computing the database was even more I/O intensive than running a match.

Interior-node recognition [697] is another memorization technique in game playing that includes game-theoretical information in form of score values to cut-off whole subtrees for interior node evaluation in $\alpha\beta$ search engines. Recognizers are only invoked, if transposition table lookups fail. To enrich the game theoretical information, material signatures are helpful. The memory access is layered. Firstly, appropriate recognizers are efficiently detected and selected before a lookup into an EDB is performed [387].

11.6 Other AI Areas

An apparent candidate for hierarchical memory exploitation is *data or information mining* [644]; the process of inferring knowledge from very large databases. *Web mining* [474] is data mining in the Internet where *intelligent internet systems* [502] consider user modeling, information source discovering and information integration. Classification and clustering in data mining links to the wide range of *machine learning* [553] techniques with decision-tree and statistical methods, neural networks, genetic algorithms, nearest neighbor search and rule induction. *Association rules* are implications of the form $X \Rightarrow I$ with I being a binary attribute. Set X has support s, if $s\%$ of all data is in X, whereas a rule $X \Rightarrow I$ has confidence c, if $c\%$ of all data that are in X also obey I. Given a set of transactions D, the problem is to generate all association rules that have user-specified minimum support and confidence.

The main association rule induction algorithm is *AIS* [18]. For fast discovery, the algorithm was improved in *Apriori*. The first pass of the algorithm

simply counts the number of occurrences of each item to determine itemsets of cardinality 1 with minimum support. In the k-th pass the itemset with $(k-1)$ elements and minimum support of phase $(k-1)$ are used to generate a candidate set, which by scanning the database yields the support of the candidate set and the k-itemset with minimum support. The running time of Apriori is $\mathcal{O}(|C| \cdot |D|)$, where $|D|$ is the size of the database and $|C|$ the total number of generated candidates. Even advanced association rule inference requires substantial processing power and main memory [757]. An example to hierarchical memory usage is a distributed rule discovery algorithm [189].

Case-based reasoning (CBR) [449] systems integrate database storage technology into knowledge representation systems. CBR systems store previous experiences (cases) in memory and in order to solve new problems, i) retrieve similar experience about similar situation from memory ii) complete or partial re-use or adapt the experience in the context of the new situation, iii) store new experience in memory. We give a few examples that are reported to explicitly use secondary memory. Parka-DB [706] is a knowledge base with a reduction in primary storage with 10% overhead in time, decreasing the load time by more than two orders of magnitude. Framer [369] is a disk-based object-oriented knowledge based system, whereas Thenetsys [609] is a semantic network system that employs secondary memory structure to transfer network nodes from the disk into main memory and vice versa.

Automated theorem proving procedures draw inferences on a set of clauses $\Gamma \rightarrow \Delta$, with Γ and Δ as multisets of atoms. A top-down proof creates a proof tree, where the node label of each interior node corresponds to the conclusion, and the node labels of its children correspond to the premises of an inference step. Leaves of the proof tree are axioms or instances of proven theorems. A *proof state* represents the outer fragment of a proof tree: the top-node, representing the goal and all leaves, representing the subgoals of the proof state. All proven leaves can be discharged, because they are not needed for further proof search. If all subgoals have been solved, the proof is successful. Similar to action planning, proof-state based automated theorem proving spans large and infinite state spaces. The overall problem is undecidable and can be tackled by user invention and implicit enumeration only. While polynomial decision procedures exists for restricted classes [538], first general heuristic search algorithms to accelerate exploration have been proposed [270].

11.7 Conclusions

The spectrum of research in memory limited algorithms for representing and exploring large or even infinite problem spaces is enormous and encompasses large subareas of AI. We have seen alternative approaches to exploit and memorize problem specific knowledge and some schemes that explicitly schedule external memory. Computational trade-offs under bounded re-

sources become increasingly important, as e.g. a recent issue of *Artificial Intelligence* [381] with articles on recursive conditioning, algorithm portfolios, anytime algorithms, continual computation, and iterative state space reduction indicates. Improved design of hierarchical memory algorithms, probably special-tailored to AI exploration, are apparently needed.

Nevertheless, there is much more research, sensibility, and transfer of results needed, as two feedbacks of German AI researchers illustrate. For the case of external algorithms, Bernhard Nebel [578] mentions that current memory sizes of 256MB up to several GB make the question of refined secondary memory access no longer that important. This argument neglects that even by larger amount of main memory the latency gap still rises, and that with current CPU speed, exploration engines often exhaust main memory in less than a few minutes.

For the case of processor performance tuning, action execution in robotics has a high frequency of 10-20 Hz, but there is almost no research in improved cache performance: Wolfram Burgard [165] reports some successes by restructuring loops in one application, but has also seen failures for hand-coded assembler inlines to beat the optimized compiler outcome in another.

The ultimative motivation for an increased research in space limitations and hierarchical memory usage in AI is its inspirator, the human brain, with an hierarchical layered organization structure, including ultra short time working memory, as well as short and long time memorization capabilities.

Acknowledgments. The work is supported by DFG in the projects *heuristic search and its application to protocol verification* and *directed model checking.*