Before Warren's work on the compilation of inference in Prolog, logic programming was too slow for general use. Compilers by Warren and others allowed Prolog code to achieve speeds that are competitive with C on a variety of standard benchmarks (Van Roy, 1990). Of course, the fact that one can write a planner or natural language parser in a few dozen lines of Prolog makes it somewhat more desirable than C for prototyping most small-scale AI research projects.

Parallelization can also provide substantial speedup. There are two principal sources of parallelism. The first, called OR-parallelism, comes from the possibility of a goal unifying with many different clauses in the knowledge base. Each gives rise to an independent branch in the search space that can lead to a potential solution, and all such branches can be solved in parallel. The second, called AND-parallelism, comes from the possibility of solving each conjunct in the body of an implication in parallel. AND-parallelism is more difficult to achieve, because solutions for the whole conjunction require consistent bindings for all the variables. Each conjunctive branch must communicate with the other branches to ensure a global solution.

### 9.4.4 Redundant inference and infinite loops

We now turn to the Achilles heel of Prolog: the mismatch between depth-first search and search trees that include repeated states and infinite paths. Consider the following logic program that decides if a path exists between two points on a directed graph:

\[
\text{path}(X, Z) \quad \text{link}(X, Z).
\]
\[
\text{path}(X, Z) \quad \text{path}(X, Y), \text{link}(Y, Z).
\]

A simple three-node graph, described by the facts `link(a, b)` and `link(b, c)`, is shown in Figure 9.9(a). With this program, the query `path(a, Z)` generates the proof tree shown in Figure 9.10(a). On the other hand, if we put the two clauses in the order

\[
\text{path}(X, Z) \quad \text{path}(X, Y), \text{link}(Y, Z).
\]
\[
\text{path}(X, Z) \quad \text{link}(X, Z).
\]

then Prolog follows the infinite path shown in Figure 9.10(b). Prolog is therefore incomplete as a theorem prover for definite clauses—even for Datalog programs, as this example shows—because, for some knowledge bases, it fails to prove sentences that are entailed. Notice that forward chaining does not suffer from this problem: once `path(a, b)` , `path(b, c)` , and `path(a, c)` are inferred, forward chaining halts.

Depth-first backward chaining also has problems with redundant computations. For example, when finding a path from A to B in Figure 9.9(b), Prolog performs 877 inferences, most of which involve finding all possible paths to nodes from which the goal is unreachable. This is similar to the repeated-state problem discussed in Chapter 3. The total amount of inference can be exponential in the number of ground facts that are generated. If we apply forward chaining instead, at most 2^n path facts can be generated linking n nodes. For the problem in Figure 9.9(b), only 62 inferences are needed.

Forward chaining on graph search problems is an example of dynamic programming, in which the solutions to subproblems are constructed incrementally from those of smaller
subproblems and are cached to avoid recomputation. We can obtain a similar effect in a backward chaining system using memoization—that is, caching solutions to subgoals as they are found and then reusing those solutions when the subgoal recurs, rather than repeating the previous computation. This is the approach taken by tabled logic programming systems, which use efficient storage and retrieval mechanisms to perform memoization. Tabled logic programming combines the goal-directedness of backward chaining with the dynamic-programming efficiency of forward chaining. It is also complete for Datalog knowledge bases, which means that the programmer need worry less about infinite loops. (It is still possible to get an infinite loop with predicates like \texttt{father(X, Y)} that refer to a potentially unbounded number of objects.)

\subsection*{9.4.5 Database semantics of Prolog}

Prolog uses database semantics, as discussed in Section 8.2.8. The unique names assumption says that every Prolog constant and every ground term refers to a distinct object, and the closed world assumption says that the only sentences that are true are those that are entailed