4. Representation and Problem Solving

4.1 Knowledge Representation and Reasoning

Knowledge representation and reasoning (KRR, KR&R, KR^2) is the field of artificial intelligence (AI) dedicated to representing information about the world in a form that a computer system can use to solve complex tasks such as diagnosing a medical condition or having a dialog in a natural language. Knowledge representation incorporates findings from psychology about how humans solve problems and represent knowledge in order to design formalisms that will make complex systems easier to design and build. Knowledge representation and reasoning also incorporates findings from logic to automate various kinds of reasoning, such as the application of rules or the relations of sets and subsets.

Examples of knowledge representation formalisms include semantic nets, systems architecture, frames, rules, and ontologies. Examples of automated reasoning engines include inference engines, theorem provers, and classifiers.

阅读后结合上课内容掌握知识表示的基本概念和意义，结合《人工智能教程第 2 章》进行理解。

4.2 Knowledge Representation History

The earliest work in computerised knowledge representation was focused on general problem-solvers such as the General Problem Solver (GPS) system developed by Allen Newell and Herbert A. Simon in 1959. These systems featured data structures for planning and decomposition. The system would begin with a goal. It would then decompose that goal into sub-goals and then set out to construct strategies that could accomplish each subgoal.

In these early days of AI, general search algorithms such as A* were also developed. However, the amorphous problem definitions for systems such as GPS meant that they worked only for very constrained toy domains (e.g. the "blocks world"). In order to tackle non-toy problems, AI researchers such as Ed Feigenbaum and Frederick Hayes-Roth realised that it was necessary to focus systems on more constrained problems.

These efforts led to the cognitive revolution in psychology and to the phase of AI focused on knowledge representation that resulted in expert systems in the 1970s and 80s, production systems, frame languages, etc. Rather than general problem solvers, AI changed its focus to expert systems that could match human competence on a specific task, such as medical diagnosis.

Expert systems gave us the terminology still in use today where AI systems are divided into a knowledge base, with facts about the world and rules, and an inference engine, which applies the rules to the knowledge base in order to answer questions and solve problems. In these early systems the knowledge base tended to be a fairly flat structure, essentially assertions about the values of variables used by the rules.

In addition to expert systems, other researchers developed the concept of frame-based languages in the mid-1980s. A frame is similar to an object class: It is an abstract description of a category describing things in the world, problems, and potential solutions. Frames were originally used on systems geared toward human interaction, e.g. understanding natural language and the social settings in which various default expectations such as ordering food in a restaurant narrow the search space and allow the system to choose appropriate responses to dynamic situations.

It was not long before the frame communities and the rule-based researchers realised that there was a synergy between their approaches. Frames were good for representing the real world, described as classes, subclasses, slots (data values) with various constraints on possible values. Rules were good for representing and utilising complex logic such as the process to make a medical diagnosis. Integrated systems were developed that combined frames and rules. One of the most powerful and well known was the 1983 Knowledge Engineering Environment (KEE) from Intellicorp. KEE had a complete rule engine with forward and backward chaining. It also had a complete frame-based knowledge base with triggers, slots (data values), inheritance, and message passing. Although message passing originated in the object-oriented community rather than AI it was quickly embraced by AI researchers as well in environments such as KEE and in the operating systems for Lisp machines from Symbolics, Xerox, and Texas Instruments.

The integration of frames, rules, and object-oriented programming was significantly driven by commercial ventures such as KEE and Symbolics spun off from various research projects. At the same time as this was occurring, there was another strain of research that was less commercially focused and was driven by mathematical logic and automated theorem proving. One of the most

influential languages in this research was the KL-ONE language of the mid-'80s. KL-ONE was a frame language that had a rigorous semantics, formal definitions for concepts such as an Is-A relation. KL-ONE and languages that were influenced by it such as Loom had an automated reasoning engine that was based on formal logic rather than on IF-THEN rules. This reasoner is called the classifier. A classifier can analyse a set of declarations and infer new assertions, for example, redefine a class to be a subclass or superclass of some other class that wasn't formally specified. In this way the classifier can function as an inference engine, deducing new facts from an existing knowledge base. The classifier can also provide consistency checking on a knowledge base (which in the case of KL-ONE languages is also referred to as an Ontology).

Another area of knowledge representation research was the problem of common sense reasoning. One of the first realisations learned from trying to make software that can function with human natural language was that humans regularly draw on an extensive foundation of knowledge about the real world that we simply take for granted but that is not at all obvious to an artificial agent. Basic principles of common sense physics, causality, intentions, etc. An example is the frame problem, that in an event driven logic there need to be axioms that state things maintain position from one moment to the next unless they are moved by some external force. In order to make a true artificial intelligence agent that can converse with humans using natural language and can process basic statements and questions about the world, it is essential to represent this kind of knowledge. One of the most ambitious programs to tackle this problem was Doug Lenat's Cyc project. Cyc established its own Frame language and had large numbers of analysts document various areas of common sense reasoning in that language. The knowledge recorded in Cyc included common sense models of time, causality, physics, intentions, and many others.

The starting point for knowledge representation is the knowledge representation hypothesis first formalised by Brian C. Smith in 1985: **Any mechanically embodied intelligent process will be comprised of structural ingredients that a) we as external observers naturally take to represent a propositional account of the knowledge that the overall process exhibits, and b) independent of such external semantic attribution, play a formal but causal and essential role in engendering the behaviour that manifests that knowledge.**

Currently, one of the most active areas of knowledge representation research are projects associated with the Semantic Web. The Semantic Web seeks to add a layer of semantics (meaning) on top of the current Internet. Rather than indexing web sites and pages via keywords, the Semantic Web creates large ontologies of concepts. Searching for a concept will be more effective than traditional text only searches. Frame languages and automatic classification play a big part in the vision for the future Semantic Web. The automatic classification gives developers technology to provide order on a constantly evolving network of knowledge. Defining ontologies that are static and in-

capable of evolving on the fly would be very limiting for Internet-based systems. The classifier technology provides the ability to deal with the dynamic environment of the Internet.

Recent projects funded primarily by the Defense Advanced Research Projects Agency (DARPA) have integrated frame languages and classifiers with markup languages based on XML. The Resource Description Framework (RDF) provides the basic capability to define classes, subclasses, and properties of objects. The Web Ontology Language (OWL) provides additional levels of semantics and enables integration with classification engines.

此部分为扩展内容，阅读后结合上课内容掌握知识表示的发展脉络。

4.3 Knowledge Representation Basics

Knowledge-representation is a field of artificial intelligence that focuses on designing computer representations that capture information about the world that can be used for solving complex problems.

The justification for knowledge representation is that conventional procedural code is not the best formalism to use to solve complex problems. Knowledge representation makes complex software easier to define and maintain than procedural code and can be used in expert systems.

For example, talking to experts in terms of business rules rather than code lessens the semantic gap between users and developers and makes development of complex systems more practical.

Knowledge representation goes hand in hand with automated reasoning because one of the main purposes of explicitly representing knowledge is to be able to reason about that knowledge, to make inferences, assert new knowledge, etc. Virtually all knowledge representation languages have a reasoning or inference engine as part of the system.

A key trade-off in the design of a knowledge representation formalism is that between expressivity and practicality. The ultimate knowledge representation formalism in terms of expressive power and compactness is First Order Logic (FOL). There is no more powerful formalism than that used by mathematicians to define general propositions about the world. However, FOL has two drawbacks as a knowledge representation formalism: ease of use and practicality of implementation. First order logic can be intimidating even for many software developers. Languages that do not have the complete formal power of FOL can still provide close to the same expressive power with a user interface that is more practical for the average developer to understand. The issue of practicality of implementation is that FOL in some ways is too expressive. With FOL it is possible to create statements (e.g. quantification over infinite sets) that would cause a system to never terminate if it attempted to verify them.

Thus, a subset of FOL can be both easier to use and more practical to implement. This was a driving motivation behind rule-based expert systems. IF-THEN rules provide a subset of FOL but a

very useful one that is also very intuitive. The history of most of the early AI knowledge representation formalisms; from databases to semantic nets to theorem provers and production systems can be viewed as various design decisions on whether to emphasise expressive power or computability and efficiency.

In a key 1993 paper on the topic, Randall Davis of MIT outlined five distinct roles to analyse a knowledge representation framework:

- A knowledge representation is most fundamentally a surrogate, a substitute for the thing itself, used to enable an entity to determine consequences by thinking rather than acting, i.e., by reasoning about the world rather than taking action in it.
- It is a set of ontological commitments, i.e., an answer to the question: In what terms should I think about the world?
- It is a fragmentary theory of intelligent reasoning, expressed in terms of three components: (i) the representation's fundamental conception of intelligent reasoning; (ii) the set of inferences the representation sanctions; and (iii) the set of inferences it recommends.
- It is a medium for pragmatically efficient computation, i.e., the computational environment in which thinking is accomplished. One contribution to this pragmatic efficiency is supplied by the guidance a representation provides for organising information so as to facilitate making the recommended inferences.
- It is a medium of human expression", i.e., a language in which we say things about the world."

Knowledge representation and reasoning are a key enabling technology for the Semantic Web. Languages based on the Frame model with automatic classification provide a layer of semantics on top of the existing Internet. Rather than searching via text strings as is typical today, it will be possible to define logical queries and find pages that map to those queries. The automated reasoning component in these systems is an engine known as the classifier. Classifiers focus on the subsumption relations in a knowledge base rather than rules. A classifier can infer new classes and dynamically change the ontology as new information becomes available. This capability is ideal for the ever-changing and evolving information space of the Internet.

The Semantic Web integrates concepts from knowledge representation and reasoning with markup languages based on XML. The Resource Description Framework (RDF) provides the basic capabilities to define knowledge-based objects on the Internet with basic features such as Is-A relations and object properties. The Web Ontology Language (OWL) adds additional semantics and integrates with automatic classification reasoners.

阅读后结合上课内容掌握知识表示基本定义和过程，结合《人工智能教程第 2 章》进行理解。

4.4 Propositional Logic

Propositional calculus is a branch of logic. It is also called propositional logic, statement logic, sentential calculus, sentential logic, or sometimes zeroth-order logic. It deals with propositions (which can be true or false) and relations between propositions, including the construction of arguments based on them. Compound propositions are formed by connecting propositions by logical connectives. Propositions that contain no logical connectives are called atomic propositions.

Unlike first-order logic, propositional logic does not deal with non-logical objects, predicates about them, or quantifiers. However, all the machinery of propositional logic is included in first-order logic and higher-order logics. In this sense, propositional logic is the foundation of first-order logic and higher-order logic.

Propositional Logic Basics

Logical connectives are found in natural languages. In English for example, some examples are "and" (conjunction), "or" (disjunction), "not" (negation) and "if" (but only when used to denote material conditional).

The following is an example of a very simple inference within the scope of propositional logic:

- Premise 1: If it's raining then it's cloudy.
- Premise 2: It's raining.
- Conclusion: It's cloudy.

Both premises and the conclusion are propositions. The premises are taken for granted, and with the application of modus ponens (an inference rule), the conclusion follows.

As propositional logic is not concerned with the structure of propositions beyond the point where they can't be decomposed any more by logical connectives, this inference can be restated replacing those atomic statements with statement letters, which are interpreted as variables representing statements:

- Premise 1: $P \rightarrow Q$
- Premise 2: P
- Conclusion: Q

The same can be stated succinctly in the following way:

$$P \rightarrow Q, P \vdash Q$$

When P is interpreted as "It's raining" and Q as "it's cloudy" the above symbolic expressions can be seen to correspond exactly with the original expression in natural language. Not only that, but they will also correspond with any other inference of this form, which will be valid on the same basis this inference is.

Propositional logic may be studied through a formal system in which formulas of a formal language may be interpreted to represent propositions. A system of axioms and inference rules allows certain formulas to be derived. These derived formulas are called theorems and may be interpreted to be true propositions. A constructed sequence of such formulas is known as a derivation or proof and the last formula of the sequence is the theorem. The derivation may be interpreted as proof of the proposition represented by the theorem.

When a formal system is used to represent formal logic, only statement letters (usually capital roman letters such as P, Q and R) are represented directly. The natural language propositions that arise when they're interpreted are outside the scope of the system, and the relation between the formal system and its interpretation is likewise outside the formal system itself.

In classical truth-functional propositional logic, formulas are interpreted as having precisely one of two possible truth values, the truth value of true or the truth value of false. The principle of bivalence and the law of excluded middle are upheld. Truth-functional propositional logic defined as such and systems isomorphic to it are considered to be zeroth-order logic. However, alternative propositional logics are also possible. For more, see Other logical calculi below.

The following outlines a standard propositional calculus. Many different formulations exist which are all more or less equivalent, but differ in the details of:

- their language (i.e., the particular collection of primitive symbols and operator symbols),
- the set of axioms, or distinguished formulas, and
- the set of inference rules.

Any given proposition may be represented with a letter called a 'propositional constant', analogous to representing a number by a letter in mathematics (e.g., a = 5). All propositions require exactly one of two truth-values: true or false. For example, let P be the proposition that it is raining outside. This will be true (P) if it is raining outside, and false otherwise $(\neg P)$.

We then define truth-functional operators, beginning with negation. $(\neg P)$. represents the negation of P, which can be thought of as the denial of P. In the example above, $(\neg P)$. expresses that it is not raining outside, or by a more standard reading: "It is not the case that it is raining outside." When P is true, $(\neg P)$. is false; and when P is false, $(\neg P)$. is true. As a result, $(\neg\neg P)$. always has the same truth-value as P.

Conjunction is a truth-functional connective which forms a proposition out of two simpler propositions, for example, P and Q. The conjunction of P and Q is written $P \wedge Q$, and expresses that each are true. We read $P \wedge Q$ as "P and Q". For any two propositions, there are four possible assignments of truth values:

- P is true and Q is true
- P is true and Q is false

- P is false and Q is true
- P is false and Q is false

The conjunction of P and Q is true in case 1, and is false otherwise. Where P is the proposition that it is raining outside and Q is the proposition that a cold-front is over Kansas, $P \wedge Q$ is true when it is raining outside and there is a cold-front over Kansas. If it is not raining outside, then $P \wedge Q$ is false; and if there is no cold-front over Kansas, then $P \wedge Q$ is also false.

Disjunction resembles conjunction in that it forms a proposition out of two simpler propositions. We write it $P \vee Q$, and it is read "P or Q". It expresses that either P or Q is true. Thus, in the cases listed above, the disjunction of P with Q is true in all cases—except case 4. Using the example above, the disjunction expresses that it is either raining outside, or there is a cold front over Kansas. (Note, this use of disjunction is supposed to resemble the use of the English word "or". However, it is most like the English inclusive "or", which can be used to express the truth of at least one of two propositions. It is not like the English exclusive "or", which expresses the truth of exactly one of two propositions. In other words, the exclusive "or" is false when both P and Q are true (case 1). An example of the exclusive or is: You may have a bagel or a pastry, but not both. Often in natural language, given the appropriate context, the addendum "but not both" is omitted—but implied. In mathematics, however, "or" is always inclusive or; if exclusive or is meant it will be specified, possibly by "xor".)

Material conditional also joins two simpler propositions, and we write $P \rightarrow Q$, which is read "if P then Q". The proposition to the left of the arrow is called the antecedent, and the proposition to the right is called the consequent. (There is no such designation for conjunction or disjunction, since they are commutative operations.) It expresses that Q is true whenever P is true. Thus $P \rightarrow Q$ is true in every case above except case 2, because this is the only case when P is true but Q is not. Using the example, if P then Q expresses that if it is raining outside, then there is a cold-front over Kansas. The material conditional is often confused with physical causation. The material conditional, however, only relates two propositions by their truth-values—which is not the relation of cause and effect. It is contentious in the literature whether the material implication represents logical causation.

Biconditional joins two simpler propositions, and we write $P \leftrightarrow Q$, which is read "P if and only if Q". It expresses that P and Q have the same truth-value, and in cases 1 and 4. 'P is true if and only if Q' is true, and is false otherwise. It is very helpful to look at the truth tables for these different operators, as well as the method of analytic tableaux.

阅读后结合上课内容掌握命题逻辑的基本概念和推理过程。

4.5 Search Problem Solving

Breadth-first search

Breadth-first search (BFS) is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. Extra memory, usually a queue, is needed to keep track of the child nodes that were encountered but not yet explored.

For example, in a chess endgame a chess engine may build the game tree from the current position by applying all possible moves, and use breadth-first search to find a win position for white. Implicit trees (such as game trees or other problem-solving trees) may be of infinite size; breadth-first search is guaranteed to find a solution node if one exists.

In contrast, (plain) depth-first search, which explores the node branch as far as possible before backtracking and expanding other nodes, may get lost in an infinite branch and never make it to the solution node. Iterative deepening depth-first search avoids the latter drawback at the price of exploring the tree's top parts over and over again. On the other hand, both depth-first algorithms get along without extra memory.

Breadth-first search can be generalised to graphs, when the start node (sometimes referred to as a 'search key') is explicitly given, and precautions are taken against following a vertex twice.

BFS and its application in finding connected components of graphs were invented in 1945 by Konrad Zuse, in his (rejected) Ph.D. thesis on the Plankalkul programming language, but this was not published until 1972. It was reinvented in 1959 by Edward F. Moore, who used it to find the shortest path out of a maze, and later developed by C. Y. Lee into a wire routing algorithm (published 1961).

Pseudocode

```
Input: A graph G and a starting vertex root of G
Output: Goal state. The parent links trace the shortest path back to root
 1  procedure BFS(G, root) is
 2      let Q be a queue
 3      label root as explored
 4      Q.enqueue(root)
 5      while Q is not empty do
 6          v := Q.dequeue()
 7          if v is the goal then
 8              return v
 9          for all edges from v to w in G.adjacentEdges(v) do
10              if w is not labeled as explored then
11                  label w as explored
12                  Q.enqueue(w)
```

Time and space complexity

The time complexity can be expressed as $O(|V|+|E|)$, since every vertex and every edge will be explored in the worst case. $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph. Note that $O(|E|)$ may vary between $O(1)$ and $O(|V|^2)$, depending on how sparse the input graph is.

When the number of vertices in the graph is known ahead of time, and additional data structures are used to determine which vertices have already been added to the queue, the space complexity can be expressed as $O(|V|)$, where $|V|$ is the number of vertices. This is in addition to the space required for the graph itself, which may vary depending on the graph representation used by an implementation of the algorithm.

When working with graphs that are too large to store explicitly (or infinite), it is more practical to describe the complexity of breadth-first search in different terms: to find the nodes that are at distance d from the start node (measured in number of edge traversals), BFS takes $O(bd+1)$ time and memory, where b is the "branching factor" of the graph (the average out-degree).

Completeness

In the analysis of algorithms, the input to breadth-first search is assumed to be a finite graph, represented as an adjacency list, adjacency matrix, or similar representation. However, in the application of graph traversal methods in artificial intelligence the input may be an implicit representation of an infinite graph. In this context, a search method is described as being complete if it is guaranteed to find a goal state if one exists. Breadth-first search is complete, but depth-first search is not. When applied to infinite graphs represented implicitly, breadth-first search will eventually find the goal state, but depth first search may get lost in parts of the graph that have no goal state and never return.

Depth-first search

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

A version of depth-first search was investigated in the 19th century by French mathematician Charles Pierre Tremaux as a strategy for solving mazes.

The time and space analysis of DFS differs according to its application area. In theoretical computer science, DFS is typically used to traverse an entire graph, and takes time $O(|V|+|E|)$, where $|V|$ is the number of vertices and $|E|$ the number of edges. This is linear in the size of the graph. In these applications it also uses space $O(|V|)$ in the worst case to store the stack of vertices on the current search path as well as the set of already-visited vertices. Thus, in this setting, the

time and space bounds are the same as for breadth-first search and the choice of which of these two algorithms to use depends less on their complexity and more on the different properties of the vertex orderings the two algorithms produce.

For applications of DFS in relation to specific domains, such as searching for solutions in artificial intelligence or web-crawling, the graph to be traversed is often either too large to visit in its entirety or infinite (DFS may suffer from non-termination). In such cases, search is only performed to a limited depth; due to limited resources, such as memory or disk space, one typically does not use data structures to keep track of the set of all previously visited vertices. When search is performed to a limited depth, the time is still linear in terms of the number of expanded vertices and edges (although this number is not the same as the size of the entire graph because some vertices may be searched more than once and others not at all) but the space complexity of this variant of DFS is only proportional to the depth limit, and as a result, is much smaller than the space needed for searching to the same depth using breadth-first search. For such applications, DFS also lends itself much better to heuristic methods for choosing a likely-looking branch. When an appropriate depth limit is not known a priori, iterative deepening depth-first search applies DFS repeatedly with a sequence of increasing limits. In the artificial intelligence mode of analysis, with a branching factor greater than one, iterative deepening increases the running time by only a constant factor over the case in which the correct depth limit is known due to the geometric growth of the number of nodes per level.

DFS may also be used to collect a sample of graph nodes. However, incomplete DFS, similarly to incomplete BFS, is biased towards nodes of high degree.

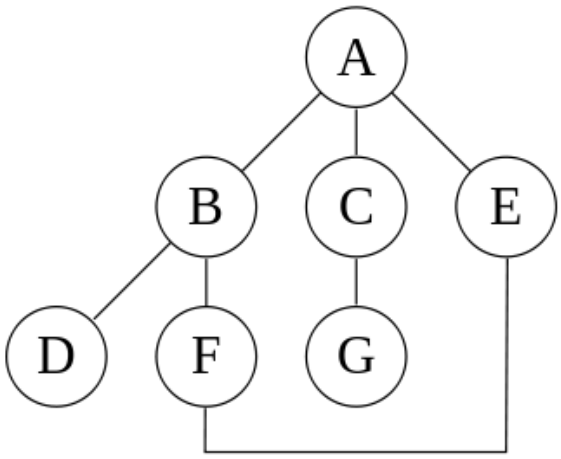
Pseudocode

```
Input: Output: A recursive implementation of DFS:

procedure DFS(G, v) is
    label v as discovered
    for all directed edges from v to w that are in G.adjacentEdges(v) do
        if vertex w is not labeled as discovered then
            recursively call DFS(G, w)
```

A non-recursive implementation of DFS with worst-case space complexity $O(|E|)$, with the possibility of duplicate vertices on the stack:

```
procedure DFS_iterative(G, v) is
    let S be a stack
    S.push(v)
    while S is not empty do
        v = S.pop()
        if v is not labeled as discovered then
```




```
            label v as discovered
            for all edges from v to w in G.adjacentEdges(v) do
                S.push(w)
```

These two variations of DFS visit the neighbours of each vertex in the opposite order from each other: the first neighbour of v visited by the recursive variation is the first one in the list of adjacent edges, while in the iterative variation the first visited neighbour is the last one in the list of adjacent edges. The recursive implementation will visit the nodes from the example graph in the following order: A, B, D, F, E, C, G. The non-recursive implementation will visit the nodes as: A, E, F, B, D, C, G.

The non-recursive implementation is similar to breadth-first search but differs from it in two ways:

1. it uses a stack instead of a queue, and

2. it delays checking whether a vertex has been discovered until the vertex is popped from the stack rather than making this check before adding the vertex.

If G is a tree, replacing the queue of the breadth-first search algorithm with a stack will yield a depth-first search algorithm. For general graphs, replacing the stack of the iterative depth-first search implementation with a queue would also produce a breadth-first search algorithm, although a somewhat nonstandard one.

Another possible implementation of iterative depth-first search uses a stack of iterators of the list of neighbours of a node, instead of a stack of nodes. This yields the same traversal as recursive DFS.

```
procedure DFS_iterative(G, v) is
    let S be a stack
    S.push(iterator of G.adjacentEdges(v))
```

```
    while S is not empty do
        if S.peek().hasNext() then
            w = S.peek().next()
            if w is not labeled as discovered then
                label w as discovered
                S.push(iterator of G.adjacentEdges(w))
        else
            S.pop()
```

Complexity

The computational complexity of DFS was investigated by John Reif. More precisely, given a graph G, let $O = (v_1, \ldots, v_n)$ be the ordering computed by the standard recursive DFS algorithm. This ordering is called the lexicographic depth-first search ordering. John Reif considered the complexity of computing the lexicographic depth-first search ordering, given a graph and a source. A decision version of the problem (testing whether some vertex u occurs before some vertex v in this order) is P-complete, meaning that it is "a nightmare for parallel processing".

A depth-first search ordering (not necessarily the lexicographic one), can be computed by a randomised parallel algorithm in the complexity class RNC. As of 1997, it remained unknown whether a depth-first traversal could be constructed by a deterministic parallel algorithm, in the complexity class NC.

Uniform-cost search

In common presentations of Dijkstra's algorithm, initially all nodes are entered into the priority queue. This is, however, not necessary: the algorithm can start with a priority queue that contains only one item, and insert new items as they are discovered (instead of doing a decrease-key, check whether the key is in the queue; if it is, decrease its key, otherwise insert it). This variant has the same worst-case bounds as the common variant, but maintains a smaller priority queue in practice, speeding up the queue operations.

Moreover, not inserting all nodes in a graph makes it possible to extend the algorithm to find the shortest path from a single source to the closest of a set of target nodes on infinite graphs or those too large to represent in memory. The resulting algorithm is called uniform-cost search (UCS) in the artificial intelligence literature and can be expressed in pseudocode as

Pseudocode

```
Input: Output: A recursive implementation of DFS:

procedure DFS(G, v) is
    label v as discovered
    for all directed edges from v to w that are in G.adjacentEdges(v) do
```

```
        if vertex w is not labeled as discovered then
            recursively call DFS(G, w)
```

Complexity

The complexity of this algorithm can be expressed in an alternative way for very large graphs: when C* is the length of the shortest path from the start node to any node satisfying the "goal" predicate, each edge has cost at least ε, and the number of neighbours per node is bounded by b, then the algorithm's worst-case time and space complexity are both in $O(b^{1+C*/\lfloor\varepsilon\rfloor})$

Further optimisations of Dijkstra's algorithm for the single-target case include bidirectional variants, goal-directed variants such as the A* algorithm, graph pruning to determine which nodes are likely to form the middle segment of shortest paths (reach-based routing), and hierarchical decompositions of the input graph that reduce s–t routing to connecting s and t to their respective "transit nodes" followed by shortest-path computation between these transit nodes using a "highway". Combinations of such techniques may be needed for optimal practical performance on specific problems.

阅读后结合上课内容掌握图搜索的基本概念和意义，对所列的宽度优先、深度优先、和统一代价搜索策略进行深入理解，结合《人工智能教程第 3 章》进行理解。