

# CS486/686: Introduction to Artificial Intelligence

## Lecture 3a - States and Uninformed Search

Jesse Hoey & Victor Zhong

School of Computer Science, University of Waterloo

January 8, 2025

Readings: Poole & Mackworth Chap. 3.1-3.5

# Searching

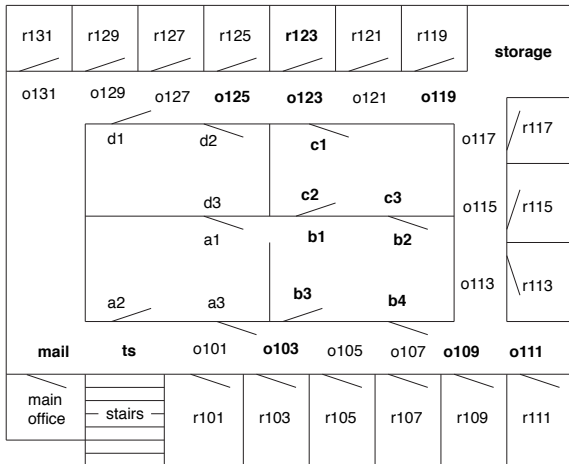
- Often we are not given an algorithm to solve a problem, but only a specification of **what is a solution**—we have to **search** for a solution
- We can do so by exploring a **directed graph** that represents the **state space** of our problem
- Sometimes the graph is literal—nodes may represent actual locations in space, and edges represent the distance between them
- Sometimes the graph is implicit—nodes represent **states**, and edges represent legal **state transitions**

# Directed Graphs

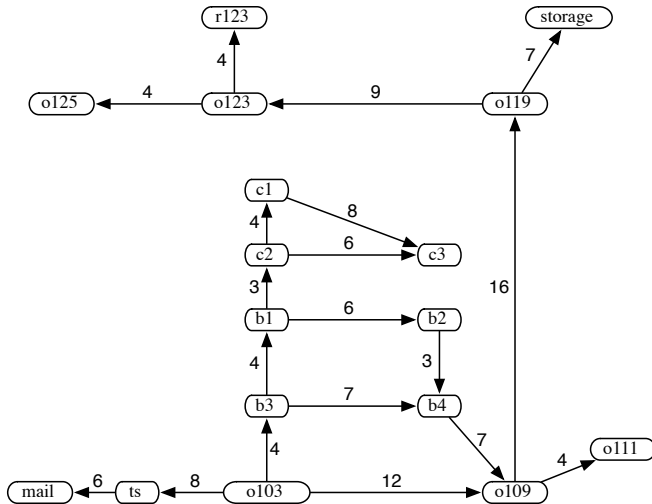
- A **graph** consists of a set  $N$  of **nodes** and a set  $A$  of ordered pairs of nodes, called **arcs** (or **edges**)
- Node  $n_2$  is a **neighbor** of  $n_1$  if there is an arc from  $n_1$  to  $n_2$  (i.e.,  $\langle n_1, n_2 \rangle \in A$ )
- A **path** is a sequence of nodes  $\langle n_0, n_1, \dots, n_k \rangle$  such that  $\langle n_{i-1}, n_i \rangle \in A$
- Often there is a **cost** associated with arcs and the cost of a path is the sum of the costs of the arcs in the path

## Example Problem for Delivery Robot

The robot wants to get from outside room 103 to the inside of room 123.



## Graph for the Delivery Robot



cost = distance travelled

# A Search Problem

## Definition (Search Problem)

A **search problem** is defined by:

- A set of **states**
- An **initial state**
- **Goal states** or a **goal test**
  - a boolean function which tells whether a given state is a goal state
- A **successor (neighbour) function**
  - an action which takes us from one state to other states
- (Optionally) a **cost** associated with each action

A solution to this problem is a path from the start state to a goal state (optionally with the smallest total cost)

## Example: 8-Puzzle

Initial State

5	3	
8	7	6
2	4	1

Goal State

1	2	3
4	5	6
7	8	

## Formulating 8-Puzzle as a Search Problem

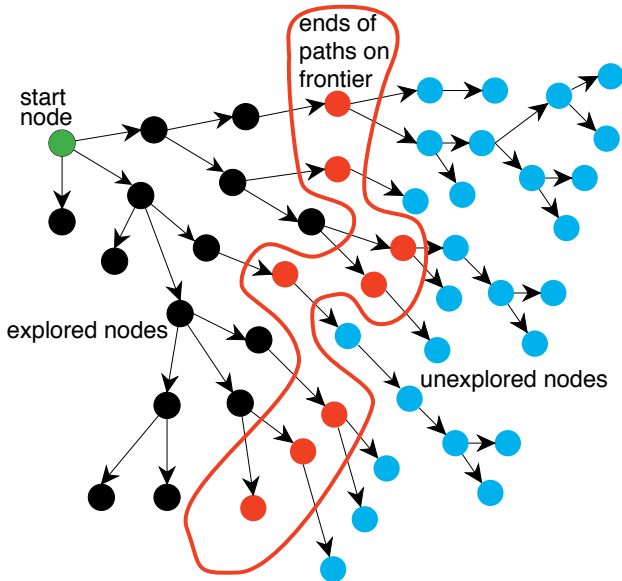
- State:  $x_{00}x_{01}x_{02}, x_{10}x_{11}x_{12}, x_{20}x_{21}x_{22}$   
 $x_{ij}$  is the number in row  $i$  and column  $j$ ,  $i, j \in \{0, 1, 2\}$   
 $x_{ij} \in \{0, \dots, 8\}$ .  $x_{ij} = 0$  denotes the empty square
- Initial state: 530, 876, 241
- Goal states: 123, 456, 780
- Successor function: Consider the empty square as a tile. State B is a successor of state A if and only if we can convert A to B by moving the empty tile up, down, left, or right by one step
- Cost function: Each move has a cost of 1



# Graph Searching

- Generic search algorithm: given a graph, start nodes, and goal nodes, **incrementally explore paths** from the start nodes
- Maintain a **frontier** of paths from the start node that have been explored
- As search proceeds, the frontier **expands** into the unexplored nodes until a goal node is encountered
- The way in which the frontier is expanded defines the **search strategy**

# Problem Solving by Graph Searching



# Graph Search Algorithm

**Input:** A graph, a set of start nodes, Boolean procedure  $\text{goal}(n)$  that tests if  $n$  is a goal node

- 1:  $\text{frontier} \leftarrow \{ \langle s \rangle : s \text{ is a start node} \}$
- 2: **while**  $\text{frontier}$  is not empty **do**
- 3:     **select** and **remove** path  $\langle n_0, \dots, n_k \rangle$  from  $\text{frontier}$
- 4:     **if**  $\text{goal}(n_k)$  **then**
- 5:         **return**  $\langle n_0, \dots, n_k \rangle$
- 6:     **for each** neighbor  $n$  of  $n_k$  **do**
- 7:         **add**  $\langle n_0, \dots, n_k, n \rangle$  to  $\text{frontier}$

# Graph Search Algorithm

- We assume that after the search algorithm returns an answer, it can be asked for more answers and the **procedure continues**
- The **neighbors** define the graph structure
- Which value is **selected** from the frontier (and how the new values are **added** to the frontier) at each stage defines the search strategy
- *Goal* defines what is a **solution**

# Types of Search

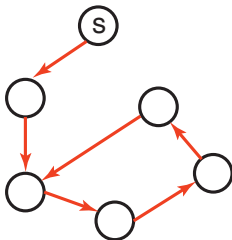
- **Uninformed (blind)**
- Heuristic
- More sophisticated “hacks”

# Depth-First Search

- **Depth-first search** treats the frontier as a **stack**
- It always selects the **last element** added to the frontier
- If the list of paths on the frontier is  $[p_1, p_2, \dots]$ 
  - $p_1$  is selected, then paths that extend  $p_1$  are added to the front of the stack (in front of  $p_2$ )
  - $p_2$  is only selected when all paths from  $p_1$  have been explored



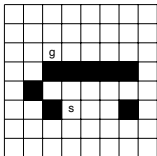
## Depth-First Search: Cycle Checking



- A searcher can **prune** a path that ends in a node **already on the path**
- We will assume that this check can be done via hashing in constant time



# Graph Search Algorithm with Cycle Check



- Use **depth-first search** to get from **s** to **g**
- Number the nodes as they are removed
- Add neighbors CW from top U,R,D,L
- Use cycle check

---

**Input:** A graph, a set of start nodes, Boolean procedure  $\text{goal}(n)$  that tests if  $n$  is a goal node

$\text{frontier} \leftarrow \{ \langle s \rangle : s \text{ is a start node} \}$

**while**  $\text{frontier}$  is not empty **do**

**select** and **remove** path  $\langle n_0, \dots, n_k \rangle$  from  $\text{frontier}$

**if**  $\text{goal}(n_k)$  **then**

**return**  $\langle n_0, \dots, n_k \rangle$

**for each** neighbor  $n$  of  $n_k$  **do**

**if**  $n \notin \langle n_0, \dots, n_k \rangle$  **then**

**add**  $\langle n_0, \dots, n_k, n \rangle$  to  $\text{frontier}$

## Recursive Implementation of DFS

An equivalent recursive implementation of DFS:

**Input:** A graph, a set of start nodes, Boolean procedure  $\text{goal}(n)$  that tests if  $n$  is a goal node

```
function DFS( $\langle n_0, \dots, n_k \rangle$ )  
  if  $\text{goal}(n_k)$  then  
    return  $\langle n_0, \dots, n_k \rangle$   
  for each neighbor  $n$  of  $n_k$  do  
    if  $n \notin \langle n_0, \dots, n_k \rangle$  then  
      DFS( $\langle n_0, \dots, n_k, n \rangle$ )  
end function  
  
for each start node  $s$  do  
  DFS( $\langle s \rangle$ )
```

The frontier is “implicitly” stored in the call stack

# Properties of DFS

## Properties

- *Space Complexity*: size of frontier in worst case
- *Time Complexity*: # nodes visited in worst case
- *Completeness*: does it find a solution when one exists?
- *Optimality*: if solution found, is it the one with the least cost?

## Useful Quantities

- $b$  is the branching factor
- $m$  is the maximum depth of the search tree
- $d$  is the depth of the shallowest goal node

## Properties of DFS: Space Complexity

$\mathcal{O}(bm)$

- $b$  is the branching factor (max number of children of any node)  
 $m$  is the max depth of the search tree
- Linear in  $m$
- Remembers  $m$  nodes on current path and at most  $b$  siblings for each node

## Properties of DFS: Time Complexity

$$\mathcal{O}(b^m)$$

- $b$  is the branching factor (max number of children of any node)  
 $m$  is the max depth of the search tree
- Exponential in  $m$
- Visit the entire search tree in the worst case

## Properties of DFS: Completeness

Is DFS guaranteed to find a solution if a solution exists?

- No
- Will get stuck in an infinite path
- An infinite path may or may not be a cycle

## Properties of DFS: Optimality

Is DFS guaranteed to return an optimal solution if it terminates?

- No
- It pays no attention to the costs and makes no guarantee on the solution's quality

# When should we use DFS?

DFS is useful when:

- Space is restricted
- Many solutions exist, perhaps with long paths

DFS is a poor method when:

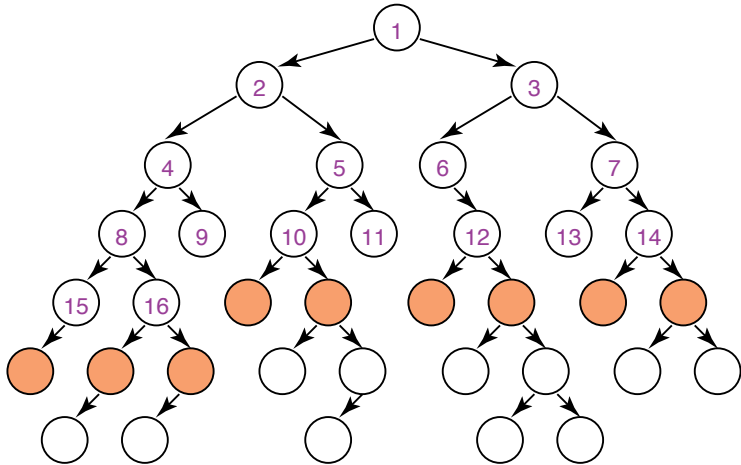
- There are infinite paths
- (Optimal) solutions are shallow
- There are multiple paths to a node



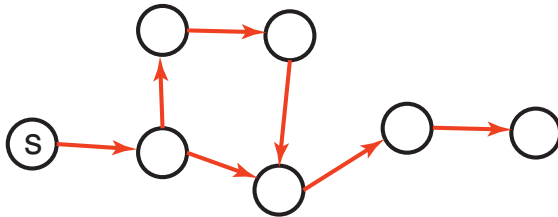
# Breadth-First Search

- **Breadth-first search** treats the frontier as a **queue**
- It always selects the **earliest** element added to the frontier
- If the list of paths on the frontier is  $[p_1, p_2, \dots, p_r]$ :
  - $p_1$  is selected, then its neighbors are added to the end of the queue, after  $p_r$
  - $p_2$  is selected next

## Illustrative Graph: Breadth-First Search

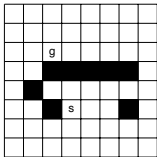


## Multiple-Path Pruning



- Multiple path pruning: **prune** a path to node  $n$  that **if any previously-found path terminates in  $n$**
- Multiple-path pruning **subsumes a cycle check** (because the current path is a path to the node)
- This entails **storing all nodes** it has found paths to
- Want to **guarantee that an optimal solution** can still be found

# Graph Search Algorithm with Multiple Path Pruning



- Use **breadth first search** to get from **s** to **g**
- Number the nodes as they are removed
- Add neighbours CW from top U,R,D,L
- Use multiple path pruning

**Input:** A graph, a set of start nodes, Boolean procedure  $\text{goal}(n)$  that tests if  $n$  is a goal node

$\text{frontier} \leftarrow \{ \langle s \rangle : s \text{ is a start node} \}$

$\text{explored} \leftarrow \emptyset$

**while**  $\text{frontier}$  is not empty **do**

**select and remove** path  $\langle n_0, \dots, n_k \rangle$  from  $\text{frontier}$

**if**  $n_k \notin \text{explored}$  **then**

$\text{explored} \leftarrow \text{explored} \cup \{ n_k \}$

**if**  $\text{goal}(n_k)$  **then**

**return**  $\langle n_0, \dots, n_k \rangle$

**for each** neighbor  $n$  of  $n_k$  **do**

**add**  $\langle n_0, \dots, n_k, n \rangle$  to  $\text{frontier}$

## Properties of BFS: Space Complexity

$$\mathcal{O}(b^d)$$

- $b$  is the branching factor  
 $d$  is the depth of the shallowest goal node
- Exponential in  $d$
- Must visit the top  $d$  levels Size of frontier is dominated by the size of level  $d$

## Properties of BFS: Time Complexity

$$\mathcal{O}(b^d)$$

- Exponential in  $d$
- Visit the entire search tree in the worst case

## Properties of BFS: Completeness

Is BFS guaranteed to find a solution if a solution exists?

- Yes
- Explores the tree level by level until it finds a goal

## Properties of BFS: Optimality

Is BFS guaranteed to return an optimal solution if it terminates?

- No
- Guaranteed to find the shallowest goal node



## When should we use BFS?

BFS is useful when:

- Space is not a concern
- We would like a solution with the fewest arcs

BFS is a poor method when:

- All the solutions are deep in the tree
- The problem is large and the graph is dynamically generated

## Combining the Best of BFS and DFS

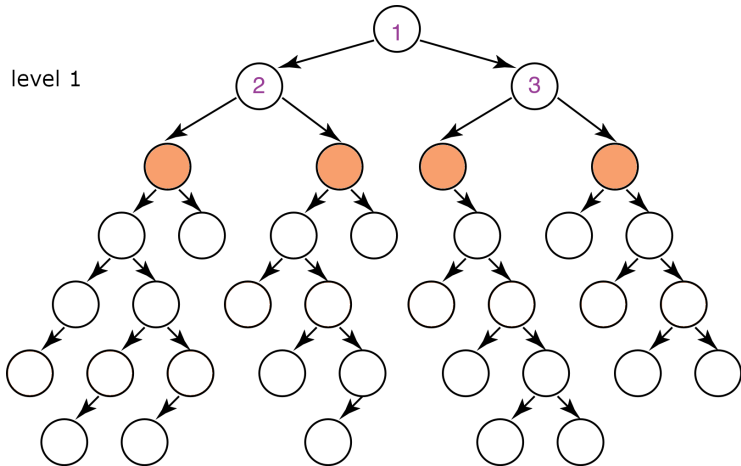
Can we create a search algorithm that combines the best of BFS and DFS?

BFS	DFS
$\mathcal{O}(b^d)$ exponential space	$\mathcal{O}(bm)$ linear space
Guaranteed to find a solution if one exists	May get stuck on infinite paths

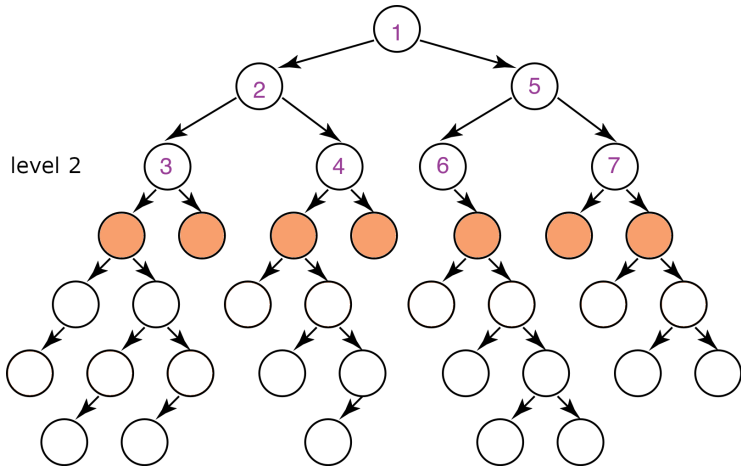
### Iterative-Deepening Search:

For every depth limit, perform depth-first search until the depth limit is reached

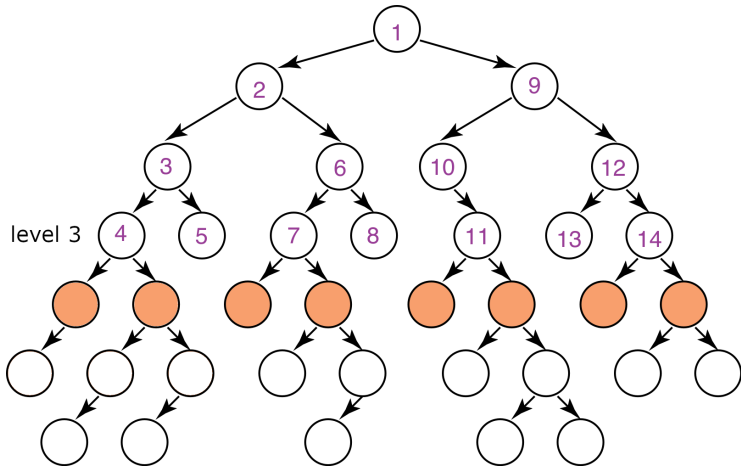
# Illustrative Graph: Iterative Deepening



## Illustrative Graph: Iterative Deepening



## Illustrative Graph: Iterative Deepening



## Properties of IDS: Space Complexity

$\mathcal{O}(bd)$

- $b$  is the branching factor  $d$  is the depth of the shallowest goal node
- Linear in  $d$  Similar to DFS
- Executes DFS for each depth limit Guaranteed to terminate at depth  $d$

## Properties of IDS: Time Complexity

$$\mathcal{O}(b^d)$$

- $b$  is the branching factor  $d$  is the depth of the shallowest goal node
- Exponential in  $d$  Similar to BFS

## Properties of IDS - Time Complexity

Complexity with solution at depth  $d$  & branching factor  $b$ :

level	# times each node is expanded		# nodes
	breadth-first	iterative deepening	
1	1	$d$	$b$
2	1	$d - 1$	$b^2$
...	...	...	...
$d - 1$	1	2	$b^{d-1}$
$d$	1	1	$b^d$
	$\geq b^d$	$\leq b^d \left(\frac{b}{b-1}\right)^2$	

$$b^d + 2b^{d-1} + 3b^{d-2} + \dots = b^d \sum_{n=1}^d n \left(\frac{1}{b}\right)^{n-1} \quad \text{rewrite} \quad (1)$$

$$< b^d \sum_{n=1}^{\infty} n \left(\frac{1}{b}\right)^{n-1} \quad \text{extend to infinity} \quad (2)$$

$$= b^d \left(\frac{b}{1-b}\right)^2 \quad \text{derivative of the geometric series} \quad (3)$$



## Properties of IDS - Completeness

Is IDS guaranteed to find a solution if a solution exists?

- Yes  
Same as BFS
- Explores the tree level by level until it finds a goal

## Properties of IDS - Optimality

Is IDS guaranteed to return an optimal solution if it terminates?

- No
- Guaranteed to find the shallowest goal node  
Same as BFS

## A Summary of IDS Properties

- Space Complexity:  $\mathcal{O}(bd)$ , linear in  $d$   
Similar to DFS
- Time Complexity:  $\mathcal{O}(b^d)$ , exponential in  $d$   
Same as BFS
- Completeness: Yes  
Same as BFS
- Optimality: No, but guaranteed to find the shallowest goal node  
Same as BFS

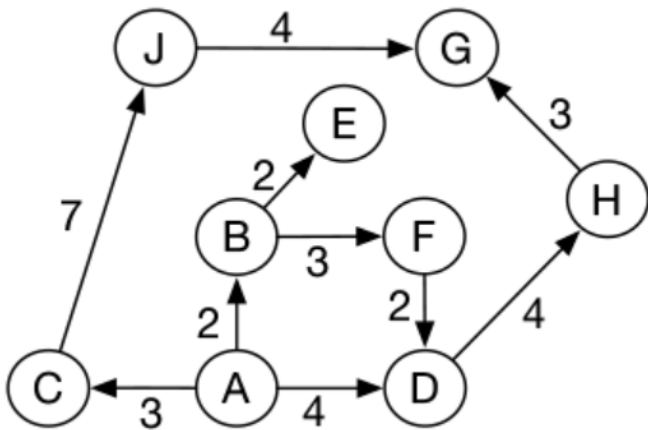
# Lowest-Cost-First Search

- Sometimes there are **costs** associated with arcs  
The cost of a path is the **sum of the costs** of its arcs

$$cost(\langle n_0, \dots, n_k \rangle) = \sum_{i=1}^k |\langle n_{i-1}, n_i \rangle|$$

- At each stage, **lowest-cost-first search** selects a path on the frontier with lowest cost
- The frontier is a **priority queue** ordered by path cost
- It finds a **least-cost path** to a goal node
- When arc costs are **equal**  $\Rightarrow$  breadth-first search
- **Uninformed/blind** search (in that it does not take the goal into account)

## Trace LCFS



How does LCFS find a path from A to G?

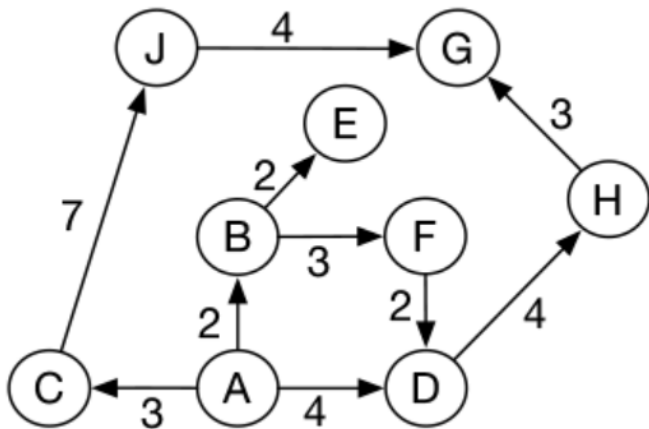
# Properties of LCFS

- Space and Time Complexities  
Both complexities are exponential  
LCFS examines a lot of paths to ensure that it returns the optimal solution first
- Completeness and Optimality  
Yes and yes under mild conditions:
  - (1) The branching factor is finite
  - (2) The cost of every edge is **strictly positive**

# Dijkstra's Algorithm

- Dijkstra's algorithm is a variant of LCFS with a kind of multiple-path pruning
- Like LCFS, the frontier is stored in a priority queue, sorted by cost
- For every node in the graph, we keep track of the lowest cost to reach it so far
- If we find a lower cost path to a node, we update that value, which may require re-sorting the priority queue
- Dijkstra's algorithm is an example of **dynamic programming** because it trades space (we must store a value for every node) for time (we may find the shortest path faster)

## Trace Dijkstra's Algorithm



How does Dijkstra's algorithm find a path from A to G?



## Next

- Informed/Heuristic Search (Poole & Mackworth Chap. 3.6-3.8)