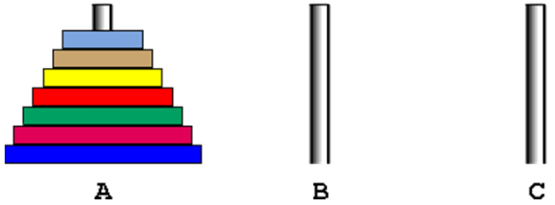
Лекция 2

Поиск в пространстве состояний:

Стратегии *неинформированного* (слепого) поиска

Головоломки – источник задач ИИ

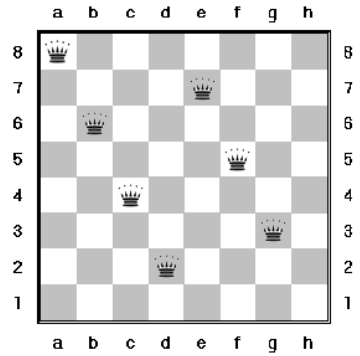
- Ханойская башня



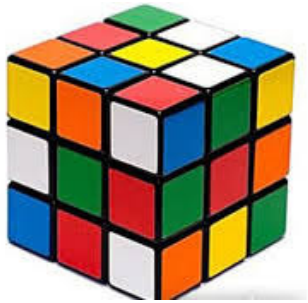
- Игра 15-ка (упрощенный вариант 8-ка)

- Задача о миссионерах и людоедах
- Задача о волке, козе и капусте

- Задача о 8 ферзях



- Кубик Рубика



- . . .

Поиск в пространстве состояний: Обобщенная формальная постановка задачи

Задача поиска в пространстве состояний задана, если задана четверка:

$$< I, \{O_i\}, GT, PC >,$$

где

I – начальное состояние (состояние мира в начале задачи);

$\{O_i\}$ – множество возможных действий (операторов перехода) для каждого состояния;

GT (*goal test*) – проверка достижения целевого состояния;

PC (*path cost*) – функция стоимости пути

Способы задания компонентов задачи поиска

Алгоритмически ориентированные способы задания компонентов задачи:

- Множество {***Oi***} операторов перехода удобно представить ***функцией последователей S*** (successor) – каждому состоянию ***x*** ставит в соответствие множество состояний ***S*(*x*)**, достижимых из ***x*** за один шаг:

  $$S : x \rightarrow S(x)$$

- Способы задания ***GT*** множества целевых состояний
  - *явное перечисление* множества целевых состояний;
  - *задание предикатов*, описывающих целевые состояния (шахматы);
- Функция *PC* (path cost) позволяет вычислить стоимость пути в заданных единицах

Задание компонентов задачи поиска

При реализации алгоритмов поиска задача представляется структурой из 4 компонентов:

- `datatype` **`Problem:`**

  `INITIAL-STATE;`

  `Operations;`

  `Goal Test;`

  `Path-Cost-Funct;`

Конкретная форма представления компонентов выбирается исходя из особенностей задачи и с учетом программной реализации

Эффективность методов поиска

- **Эффективность** *методов поиска* характеризуется следующими факторами:

  - ***полнота*** – гарантирует ли данный метод нахождение решение в принципе (при условии, что оно существует);

  - ***оптимальность*** – способность находить решение минимальной стоимости

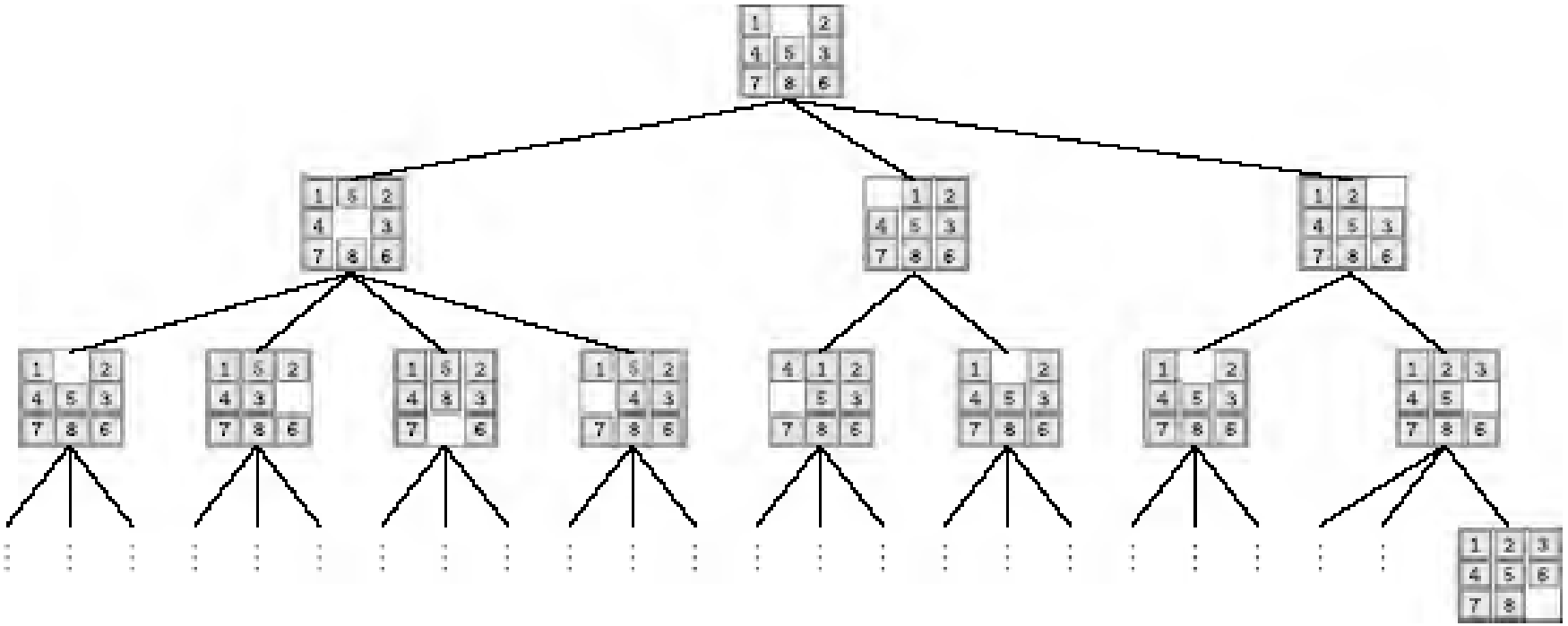
  - ***стоимость найденных решений*** – «хорошим» считается решение, имеющее *низкую стоимость пути* из начального состояния в целевое;

  - ***стоимость реализации поиска:***
  - *временная сложность* алгоритма поиска
  - *емкостная сложность* алгоритма поиска

- **Полная стоимость** *поиска* = ***стоимость пути*** + ***стоимость поиска пути***

Представление поиска деревом

- **Процесс** поиска в пространстве состояний удобно представить как процесс ***построения дерева поиска***, которое накладывается на пространство состояний.



Вершины и состояния

В задачах поиска следует различать *состояния* и *вершины.*

Состояние – элемент пространства состояний, т.е. некоторое *состояние мира* в рамках рассматриваемой задачи

Вершина – структура данных, используемая для представления дерева поиска

Вершина характеризуется:

- ***состоянием*** (`State`) в пространстве состояний, сопоставленным данной вершине;
- ***родительской вершиной*** (`Parent-Node`) – вершиной, непосредственным потомком которой является данная вершина;
- ***оператором*** (`Action`), в результате применения которого была порождена данная вершина;
- ***глубиной вершины*** (`Depth`) – числом вершин в пути от корня дерева к данной вершине;
- ***стоимостью пути*** (`Path-Cost`) от корневой вершины к текущей

Обобщенный алгоритм поиска

function *General-Search* (Problem, Strategy) **returns** solution or failure

// Инициализация дерева поиска начальным состоянием задачи

While (true) *// основной цикл*

if (нет вершин - кандидатов для раскрытия)

then return failure *// решение не найдено !*

else

выбрать в соответствии со стратегией терминальную вершину (лист) для раскрытия;

if (вершина содержит целевое состояние)

then return solution (путь к этой вершине)

else

раскрыть вершину и добавить новые вершины в дерево поиска;

end

Кайма и стратегия поиска

- ***Кайма*** (*fringer*, граница) - множество вершин, ожидающих раскрытия. В дереве поиска – множество терминальных вершин (листьев)
- ***Стратегия поиска*** - функция, выбирающая из каймы очередную вершину для раскрытия
  - стратегия может быть реализована как на этапе добавления вершин в кайму (очередь ожидающих раскрытия), так и на этапе выборки из нее

Функции и структуры для реализации обобщенного поиска

Для обобщенного описания алгоритмов поиска будем использовать следующие обозначения:

Make-Node(State) – создание вершины *для заданного состояния*;

Make-Queue(Elements) – создание очереди вершин, ожидающих раскрытия (каймы);

nodes – очередь вершин, ожидающих раскрытия (кайма);

empty(Queue) – проверка очереди на пустоту (возвращает **true**, если очередь пуста;

Remove-Front(Queue) – возвращает первый элемент очереди Queue и удаляет его из очереди;

Goal-Test(State) – проверка состояния на соответствие целевому;

Expand(node, Operators) – раскрывает вершину node, т.е. генерирует множество ее последователей с использованием операторов **Operators**;

Queueing-Fn(Queue, Elements) – добавляет в очередь Queue множество элементов Elements;

Формализованный обобщенный алгоритм поиска

function *General-Search*(*problem*, **Queuing-Fn**) **returns** solution or failure

nodes ← Make-Queue(Make-Node(*problem.*Init-State) // создаем кайму

while (true)

if (empty(*nodes*)) **then** return failure; // **нет решения !!!**

node ← Remove-Front(*nodes*);

if (*problem.*Goal-Test(*node.*State) **then** return Solution(*node*); // **РЕШЕНИЕ!!!**

nodes ← Queueing-Fn(*nodes*, Expand(*node*, *problem.*OPERATORS))

end while

end

Функция построения очереди **Queueing-Fn** определяет используемую *стратегию поиска*

Стратегии поиска

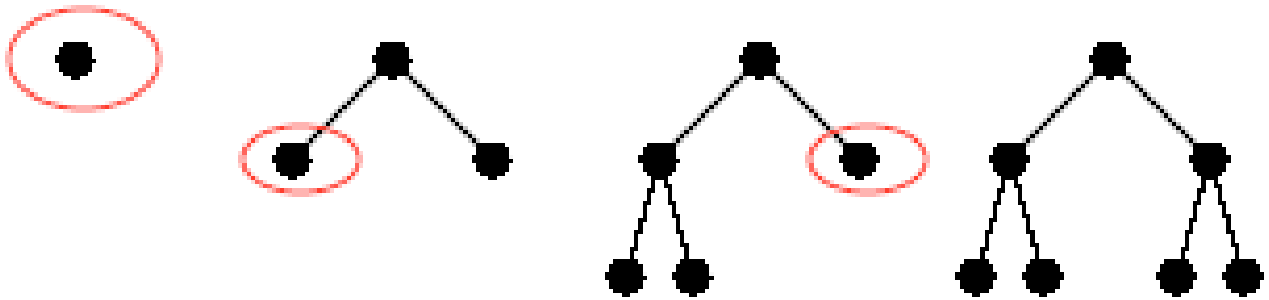
- Две группы стратегий поиска:
  - Стратегии *неинформированного* (слепого) поиска
  - Стратегии *информированного* поиска (эвристический поиск)

Стратегии неинформированного (слепого) поиска

- (Сначала) в ширину
- По критерию стоимости (Uniform-Cost Search)
- (Сначала) в глубину
- Ограниченный по глубине (Depth-Limited Search)
- С итеративным углублением (Iterative Deepening Search)
- Двунаправленный (Bi-Directional Search)

Поиск (сначала) в ширину

- **Поиск (сначала) в ширину** – все вершины глубины d раскрываются раньше вершин глубины (d+1)



- Для реализации стратегии *функция построения очереди* вершин, ожидающих раскрытия, должна ***помещать вновь сгенерированные вершины в конец очереди***:

function *Breadth-First-Search*(*problem*) **returns** solution or failure

return *General-Search*(*problem*, Enqueue-At-End)

Поиск (сначала) в ширину

Достоинства:

– ***полнота*** **-** гарантируется нахождение решения, если оно существует;

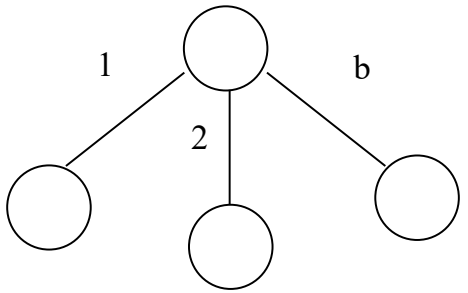
– первым всегда находится ***решение минимальной глубины***

Недостатки:

– минимальной глубине *не всегда соответствует минимальная стоимость*;

– большая сложность.

Обозначим **b** – число вершин-последователей (branching factor)



Если решение лежит на глубине **d**, для нахождения решения требуется раскрыть $1 + b + b^2 + b^3 + \ldots + b^d$ вершин

Временная и емкостная сложность поиска – $\boldsymbol{O}(b^d)$

Поиск в ширину. Оценка временной и емкостной сложности

Пусть коэффициент ветвления b=10, глубина - d.

Полагаем, что за 1 секунду проверяется 1000 вершин

d	Вершины	Время	Память
0	**1**	**1 мс**	**100 байт**
2	**111**	**0,1 с**	**11 кбайт**
4	**11111**	**11 с**	**1 Мбайт**
6	**10^6**	**18 м**	**111 Мбайт**
8	**10^8**	**314 ч**	**11 Гбайт**
10	**10^{10}**	**128 дн**	**1 Тбайт**
12	**10^{12}**	**35 лет**	**111 Тбайт**
14	**10^{14}**	**3500 лет**	**11111 Тбайт**

Поиск по критерию стоимости

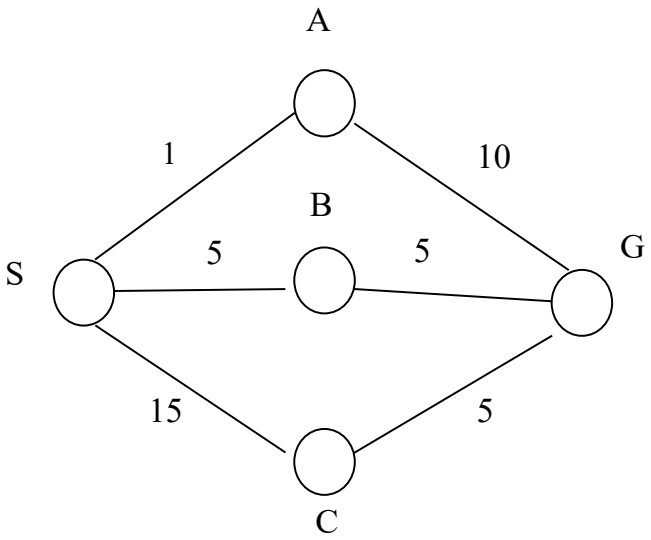
Поиск по критерию стоимости (Uniform-Cost Search) – модификация стратегии поиска в ширину, при которой ***всегда раскрывается вершина с минимальной стоимостью пути***

Пусть $g(n)$ – функция стоимости пути в вершину n.

Поиск в ширину является частным случаем поиска по критерию стоимости, когда стоимость равна глубине:

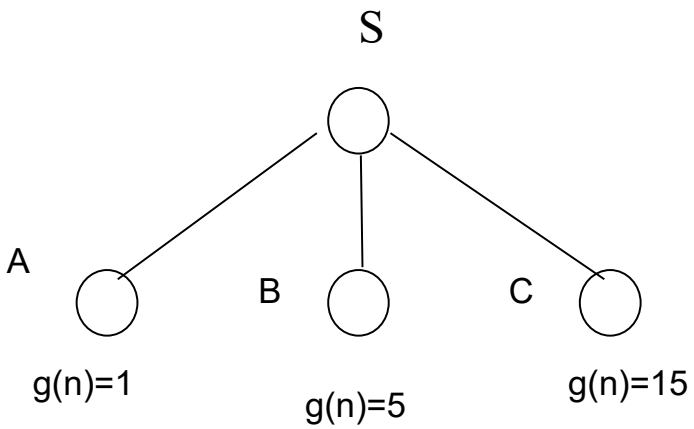
$g(n) = depth(n)$

Поиск по критерию стоимости. Пример



Найти путь минимальной стоимости из S в G.

1. Раскрыли первый ярус
2. Раскрыли A: (G) = 1+10=11
3. Проверка наличия нераскрытых вершин, у которых стоимость пути меньше полученного значения – это B.
4. Находим путь B→G: g(n) = 5+5=10 – более хорошее решение



Поиск по критерию стоимости *находит оптимальное решение*, т. е. путь минимальной стоимости, *если стоимость пути не может уменьшаться* в процессе движения вдоль пути:

g(successor(n)) >=g(n)

Поиск (сначала) в глубину

- Поиск сначала в глубину – раскрывает одну из вершин *на самом глубоком уровне дерева*

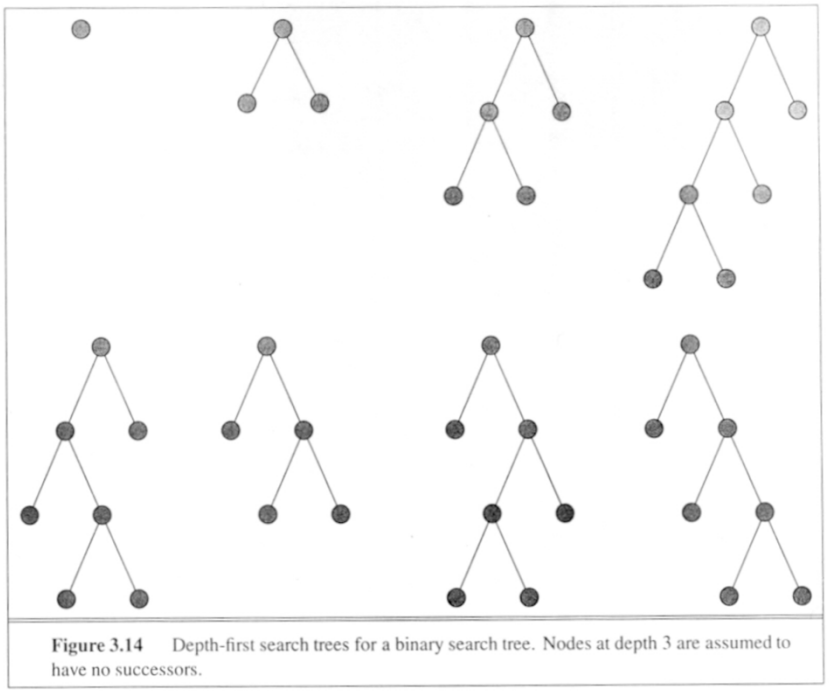


Figure 3.14 Depth-first search trees for a binary search tree. Nodes at depth 3 are assumed to have no successors.

Останавливается, когда достигнута цель или заходит в тупик – ни одна вершина нижнего уровня не может быть раскрыта.

В последнем случае выполняется возврат назад и раскрываются вершины на более верхних уровнях

- Реализация поиска *в глубину* через обобщенный поиск:

function *Depth-First-Search*(*problem*) **returns** solution or failure

return General-Search(problem, **Enqueue-At-Front**)

Поиск (сначала) в глубину

- Временная сложность – $O(b^m)$
- Емкостная сложность – $O(b*m)$, где **m –** максимальная глубина пространства поиска

  - поскольку нужно хранить только один путь из корня в вершину и множество ждущих раскрытия вершин (при d=12 поиск в ширину - **111 Тбайт**, поиск в глубину – **12 Кбайт**)
- Недостатки поиска в глубину:

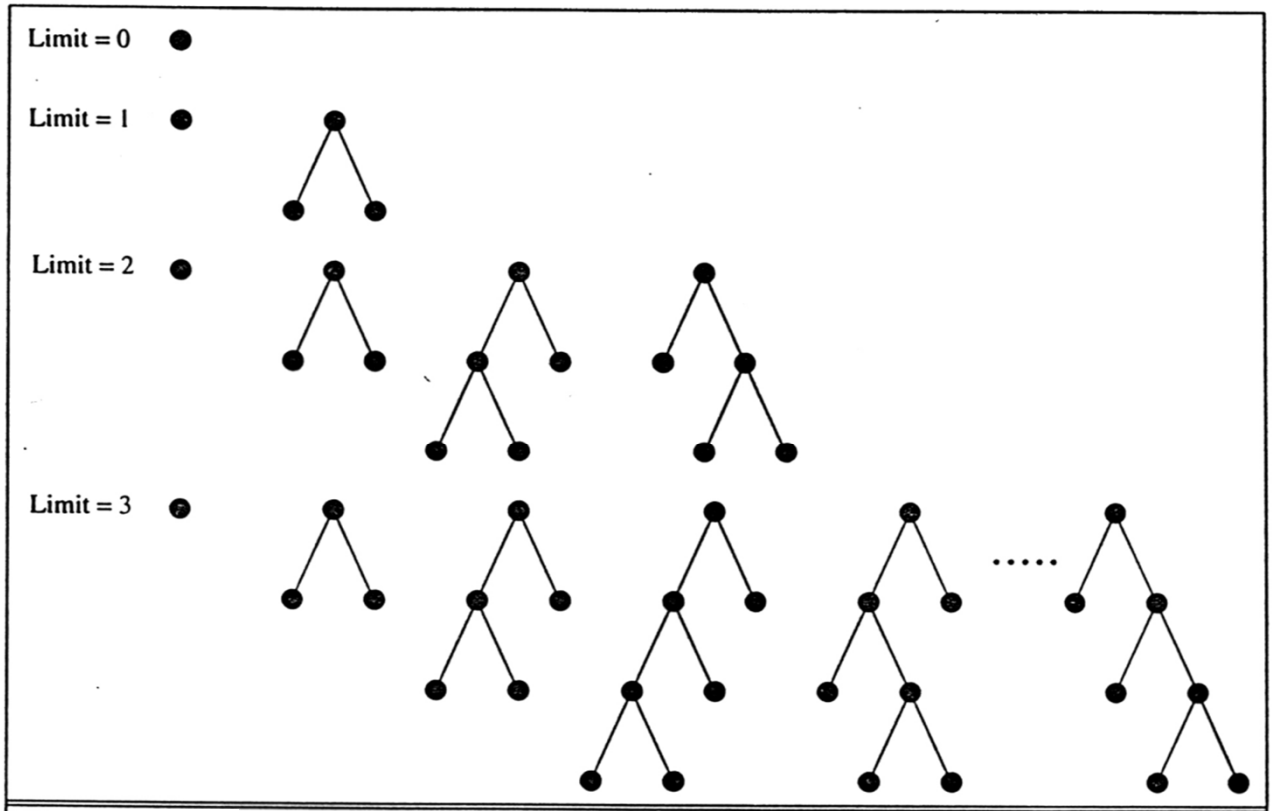
  - неполнота (может углубляться в неверном направлении, $m = \infty$. Многие задачи имеют очень глубокие или даже бесконечные деревья поиска.)
  - неоптимальность
- Эффективен, когда задача имеет много решений, т.к. повышается вероятность найти решение, исследовав малую часть пространства поиска
- Обычно реализуется с помощью рекурсивной функции

Ограниченный по глубине поиск

- Ограниченный по глубине поиск – поиск в глубину, при котором *накладываются ограничения на максимальную глубину пути* (чтобы избежать недостатков поиска в глубину)
- Сложность аналогична поиску в глубину:
  - временная – $O(b^L)$, где L – ограничение глубины
  - емкостная – $O(b*L)$
- Поиск:
  - не оптимален (даже если продолжать поиск после нахождения первого решения, т.к.оптимальное решение может находится на глубине больше L)
  - в общем случае неполон, если выбрано малое ограничение глубины
- *Диаметр пространства состояний* – максимальная длина пути для пары состояний в пространстве состояний. Если *диаметр пространства состояний* известен, он является наилучшим ограничением глубины в поиске с ограничениями по глубине.

Поиск с итеративным углублением

- Поиск с итеративным углублением (Iterative Deepening Search) – ограниченный по глубине поиск, при котором *ограничение глубины итеративно наращивается*



Four iterations of iterative deepening search on a binary tree.

Поиск с итеративным углублением

- Реализация поиска *с итеративным углублением* с использованием *ограниченного по глубине* поиска:

function *Iterative-Deepening-Search*(*problem*) **returns** solution or failure

  for depth ← 0 to ∞ do

    if *Depth-Limited-Search*(*problem*, depth) successed **then return** solution

  end

return failure

Поиск с итеративным углублением

- Сложность:
  - временная – $O(b^d)$, d – минимальная глубина решения
  - емкостная $O(b*d)$

Общее число раскрытий вершин:

$(d+1)*1 + d*b + (d-1)*b^2 + \ldots + 3*b^{d-2} + 2*b^{d-1} + b^d$ (вершины на нижнем уровне раскрываются один раз, на предпоследнем уровне - дважды и т.д., корень дерева раскрывается d+1 раз)

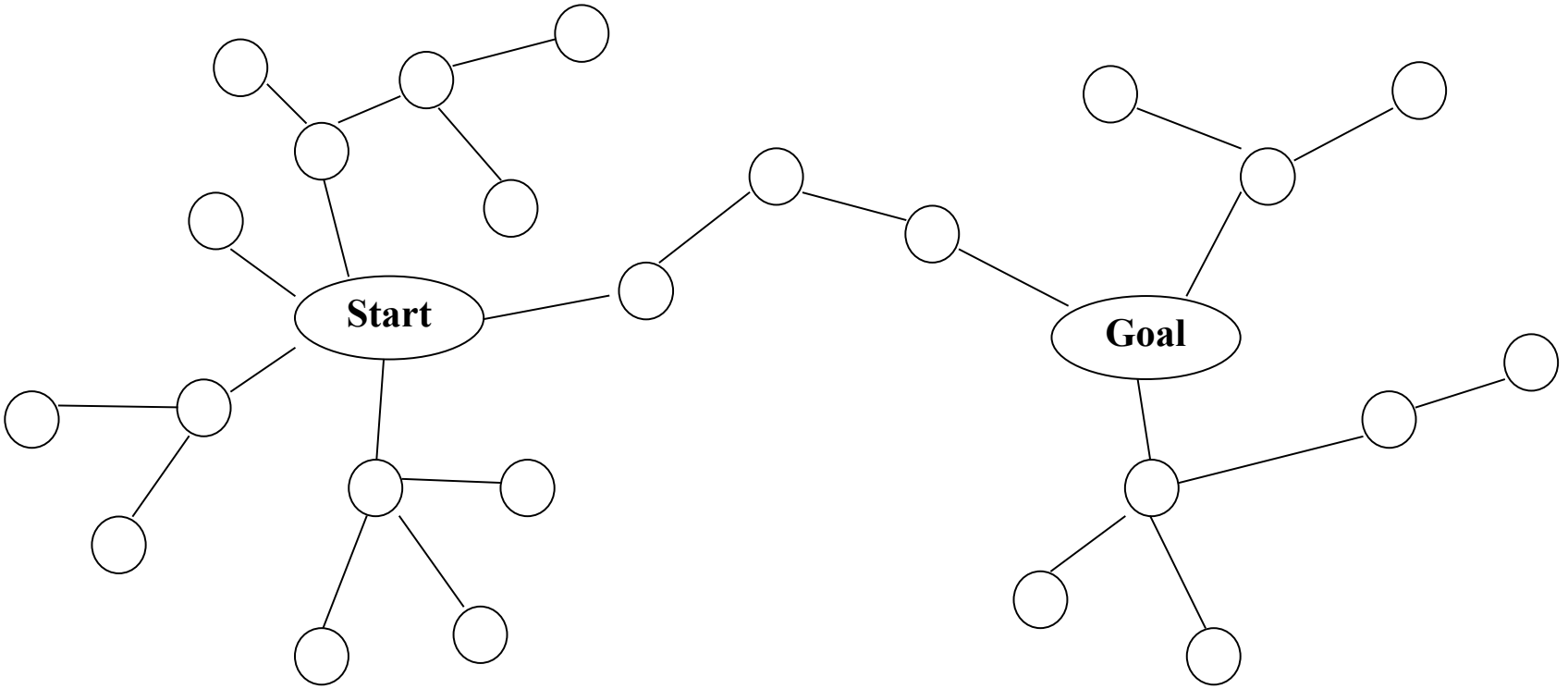
При b=10, d=5 число раскрытий = **123456**

При поиске в ширину = **111111**, т.е. **только на 11% больше!** (чем больше коэффициент ветвления, тем меньше доп. расходы от повторного раскрытия состояний)

- Поиск:
  - полон и оптимален
  - имеет умеренные требования по памяти (как поиск в глубину)
  - эффективен при большом пространстве поиска и неизвестной глубине решения
  - менее эффективен при малом коэффициенте ветвления

Двунаправленный (Bi-Directional Search)

- Двунаправленный (Bi-Directional Search) поиск – выполняется встречно от исходного состояния и целевого состояний



Двунаправленный (Bi-Directional Search)

- Двунаправленный (Bi-Directional Search) поиск – выполняется встречно от исходного и целевого состояний
- Сложность:
  - временная – $\boldsymbol{O}(2*b^{d/2})=O(b^{d/2})$, d – глубина решения
  - емкостная – $\boldsymbol{O}(b^{d/2})$
- При b=10 и d=6 *поиск в ширину* требует **1111111** шагов, *двунаправленный поиск* с каждой стороны пройдет на глубину 3, и сгенерирует **2222** вершин
- Поиск:
  - полон и оптимален;
  - требуется определить *функцию*, *определяющую предшественников* для каждого состояния;
  - требуется эффективной способ проверки нахождения вершины, достигнутой в дереве другой половины поиска
  - требуется выбрать вид поиска в каждой половине двунаправленного поиска (лучше – поиск в ширину).

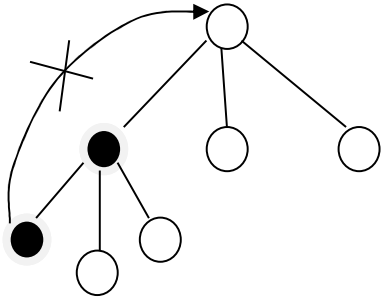
Сравнение методов поиска

Критерий	В ширину	По критерию стоимости	В глубину	Ограниченный по глубине	С итеративным углублением	Двунапра-вленный
Временная сложность	b^d	b^d	b^m	b^L	b^d	$b^{d/2}$
Емкостная сложность	b^d	b^d	$b*m$	$b*L$	$b*d$	$b^{d/2}$
Оптимальность	+	+	-	-	+	+
Полнота	+	+	-	+ если L>=d	+	+

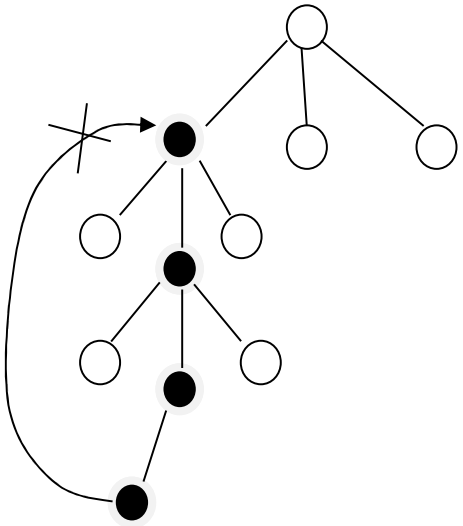
Проблема зацикливания при поиске

Три способа исключить зацикливания (повторяющиеся состояния):

1. Исключить попадание в состояние, из которого только что вышли:
   - функция раскрытия должна блокировать генерацию потомка в дереве поиска решений, если его состояние совпадает с состоянием родителя



2. Исключить циклы в дереве поиска:
   - функция раскрытия вершин должна блокировать генерацию последователя, состояние которого совпадает с состоянием любого предка данной вершины.



3. Блокировать генерацию состояний, ранее сгенерированных в дереве поиска. Сложность $O(b^d)$, d – глубина.