Содержание лекции

1. Поиск в глубину
2. Поиск в ширину
3. A*Seacrh

Деревья поиска

Игры с единственным игроком подобны деревьям игр:

- имеется начальное состояние - верхний узел в дереве поиска;
- имеется последовательность ходов, которая изменяет позицию на доске до тех пор, пока не будет достигнуто целевое состояние.

Дерево поиска

представляет набор промежуточных состояний доски в процессе выполнения алгоритма поиска пути.

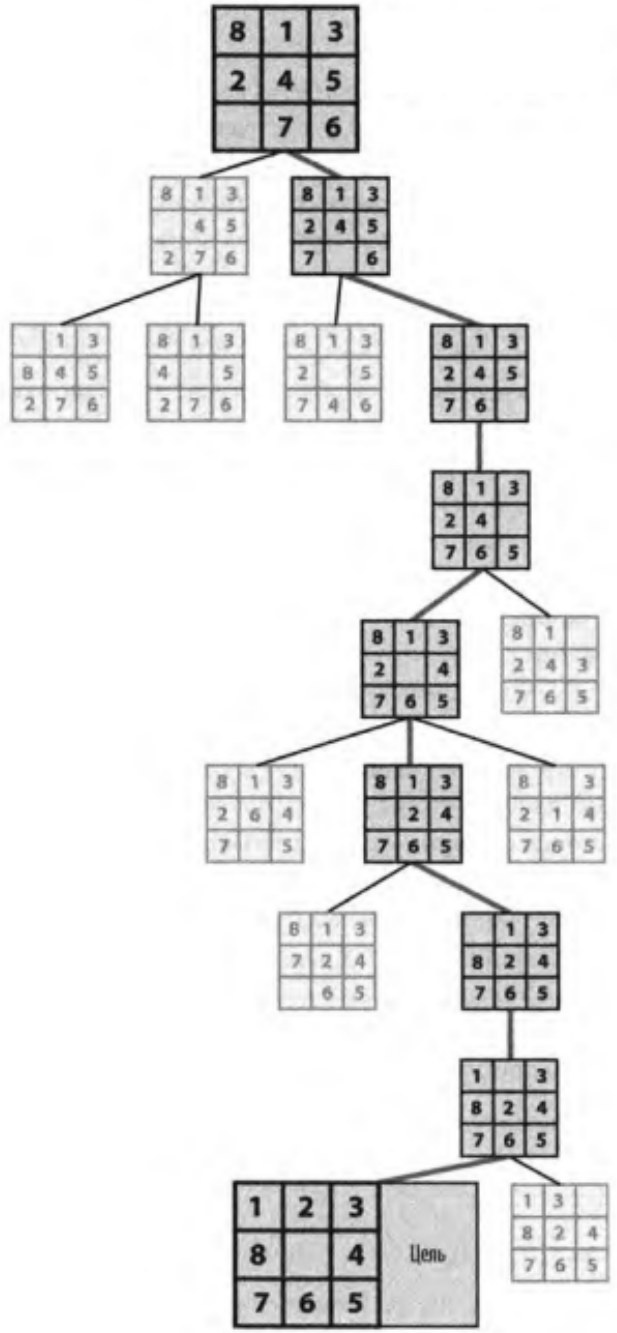
Структура данных для дерева поиска

представляет собой дерево, так как алгоритм гарантирует, что он не посещает никакое состояние доски дважды.

Цель алгоритма

определеить, в каком порядке следует посещать состояния доски в попытках достичь целевого состояния.

1. головоломка «8», играется на доске 3x3,
2. восемь квадратных плиток, пронумерованных от 1 до 8, и пустое пространство, в котором плитки нет.
3. соседняя (по горизонтали или вертикали) клитка может быть перемещена путем сдвига в пустое пространство;
4. цель: начав с начального состояния, перемещая плитки, достичь целевого состояния.



Алгоритмы поиска

Слепой поисковый алгоритм использует фиксированную стратегию вместо вычисления состояния доски.

Слепой поиск в глубину

просто играет в игру, произвольно выбирая очередной ход из доступных вариантов для данного состояния доски и выполняя откат при достижении максимальной глубины расширения.

Слепой поиск в ширину

методично исследует все возможные решения с k ходами перед тем как испытать любое решение с $k + 1$ ходами.

Алгоритм A*Search

выполняется поиск решения с использованием различных эвристических функций.

Эвристические функции

Эвристические функции

оценивают количество оставшихся ходов до целевого состояния из данного и могут использоваться для непосредственного поиска пути.

Например, в головоломке «8» такая функция будет оценивать для каждой плитки в состоянии доски число ходов, необходимых для ее размещения в нужном месте в целевом состоянии.
Наибольшая трудность в поиске пути состоит в разработке эффективной эвристической функции.

Поиск в глубину

Поиск в глубину пытается найти путь к целевому состоянию доски, делая столько ходов вперед, сколько возможно.

- некоторые деревья поиска содержат очень большое количество состояний доски, поиск в глубину оказывается практичным, только если максимальная глубина поиска фиксируется заранее;
- для того, чтобы избежать зацикливания, необходимо запоминать уже рассмотренные состояния;
- состояния, которые еще не посещены, - **открытые состояния**, помещаются в стек;
- множество посещенных состояний - закрытые состояния доски.

Алгоритм возвращает **последовательность ходов**, которая представляет собой путь от начального состояния к целевому (или сообщает, что такое решение не найдено).

Поиск в глубину

1. На каждой итерации поиск в глубину снимает со стека непосещенное состояние доски и расширяет его путем вычисления множества последующих состояний доски с учетом доступных разрешенных ходов.
2. Поиск прекращается, когда достигнуто целевое состояние доски.
3. Любой преемник состояния доски, который уже имеется во множестве закрытых состояний, отбрасывается.
4. Оставшиеся непосещенные состояния доски помещаются в стек открытых состояний, и поиск продолжается.

Поиск в глубину

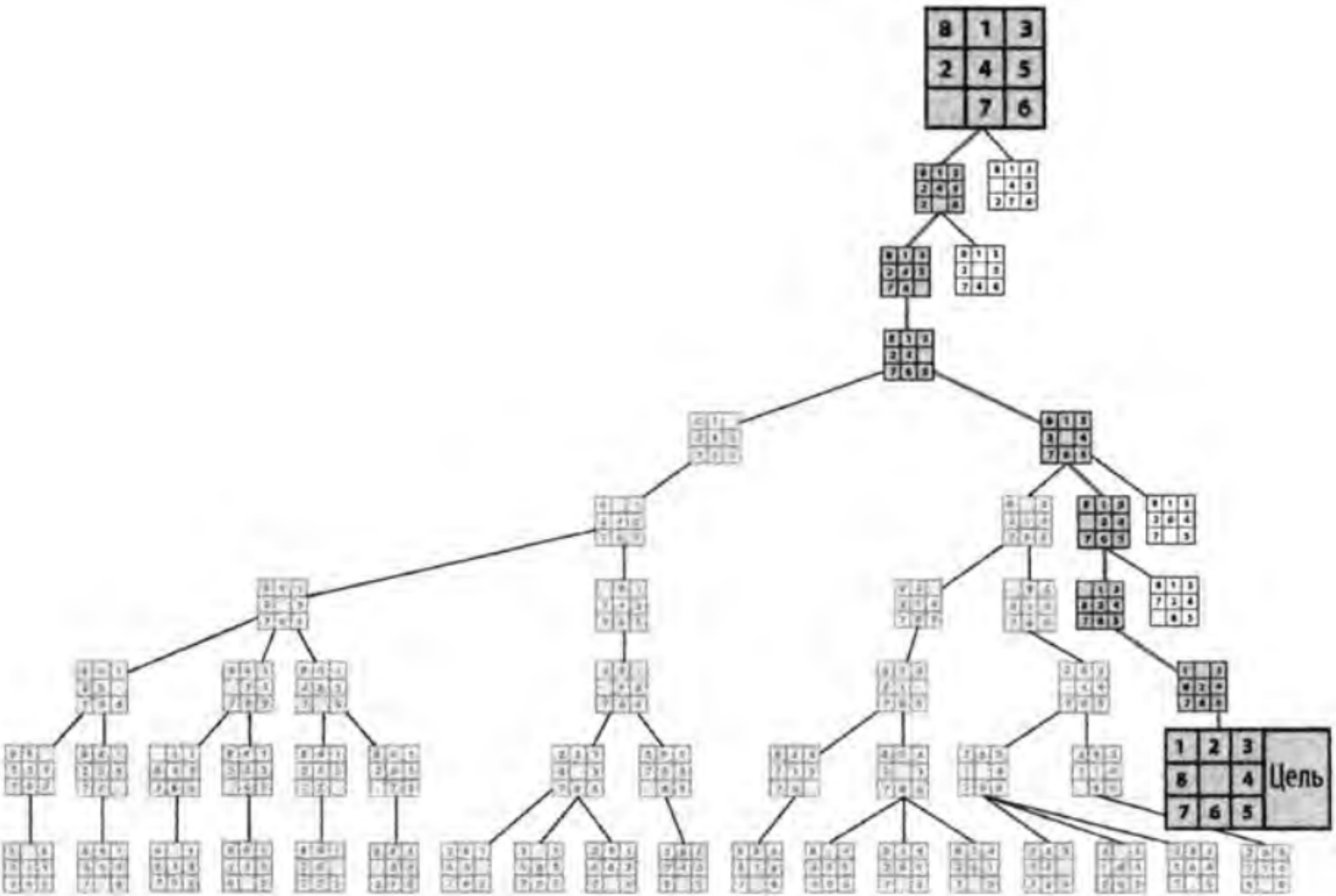
Наилучший случай: $O(b \cdot d)$; средний и наихудший случаи: $O(b^d)$

```
search (initial, goal, maxDepth)
    if initial = goal then return "Решено"
    initial.depth = 0
    open = new Stack                                          ❶
    closed = new Set
    insert (open, copy(initial))

    while open не пуст do
        n = pop (open)                                        ❷
        insert (closed, n)
        foreach допустимый ход m в n do
            nextState = состояние после хода m в n
            if closed не содержит nextState then
                nextState.depth = n.depth + 1                 ❸
                if nextState = goal then return "Решено"
                if nextState.depth < maxDepth then
                    insert (open, nextState)                  ❹

    return "Решения нет"
end
```

❶ Поиск в глубину использует стек для хранения открытых состояний, которые должны быть посещены.
❷ Последнее состояние снимается со стека.
❸ Поиск в глубину вычисляет глубину, чтобы не превысить максимальную глубину поиска.
❹ Вставка очередного состояния представляет собой операцию внесения в стек.


Цель

Анализ алгоритма поиска в глубину

Пусть d - максимальная глубина, ограничивающая поиск в глубину, а b — коэффициент ветвления дерева поиска.

Производительность алгоритма определяется:

- общими характеристиками (базовые операции);
- специфичными операции для конкретной задачи.

В общем случае замедлить работу алгоритма могут базовые операции:

- Удаление очередного состояния доски для оценки.
- Добавление состояния доски во множество закрытых состояний.
- Определение наличия состояния во множестве закрытых состояний.
- Добавление состояния доски во множество открытых состояний для позднейшего посещения.

Анализ алгоритма поиска в глубину

Специфическими для конкретной задачи характеристиками, которые влияют на производительность, являются:

- количество состояний-преемников для отдельного состояния доски;
- упорядочение допустимых ходов.

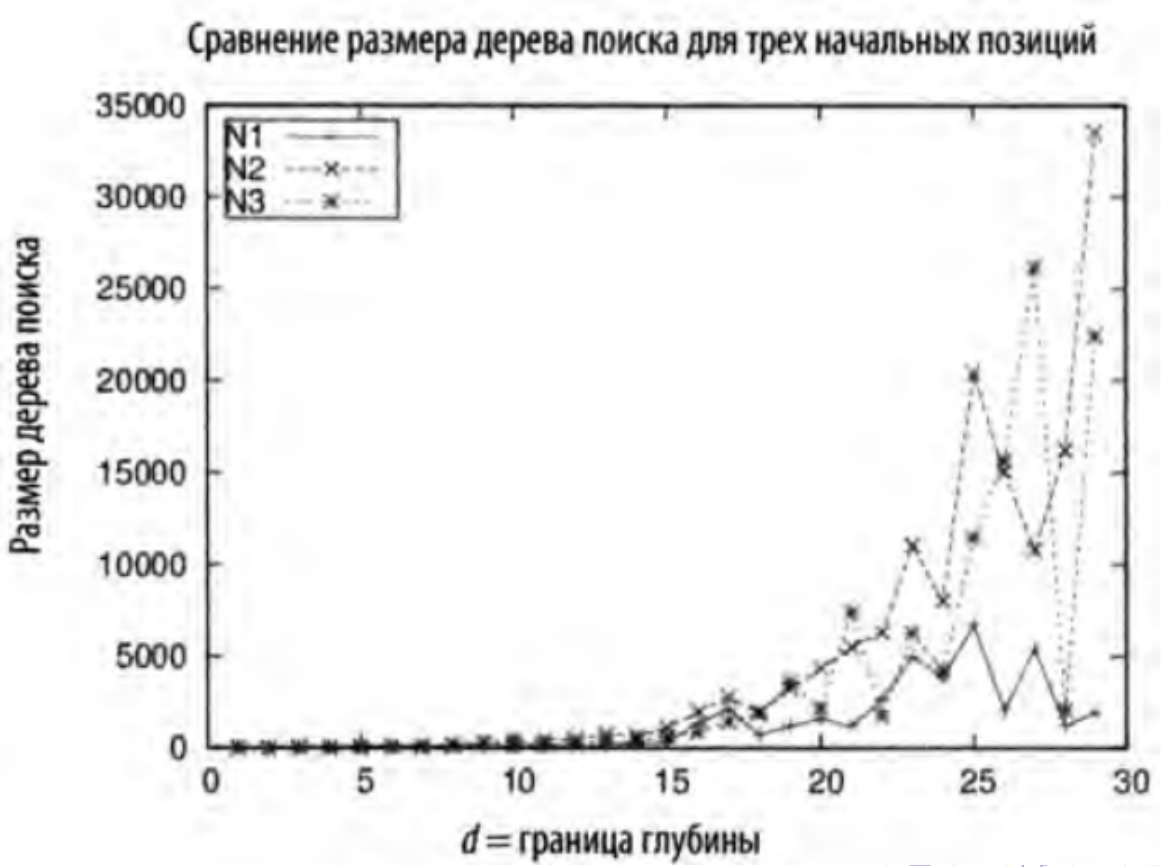
Если имеется какая-то эвристическая информация (какие ходы вероятнее других ведут к решению), то в упорядоченном списке допустимых ходов они должны появляться раньше прочих.

- В общем случае размер дерева поиска растет экспоненциально с основанием, равным коэффициенту ветвления b.
- У головоломки «8» коэффициент ветвления находится между 2 и 4, в зависимости от местоположения пустой плитки, и в среднем равен 2.67.

Начальная позиция N2

1	2	3
7	8	4
6		5

Размер деревьев поиска для поиска в глубину с увеличением предельной глубины поиска:



Поиск в ширину

Поиск в ширину пытается найти путь с помощью методичной оценки состояний доски, находящихся ближе всего к начальному состоянию.

- Кардинальным отличием от поиска в глубину является то, что поиск в ширину поддерживает очередь открытых и еще непосещенных состояний, в то время как поиск в глубину использует стек.
- На каждой итерации поиск в ширину удаляет из очереди очередное непосещенное состояние и расширяет его для вычисления множества состояний-преемников с учетом допустимых ходов.
- Если достигается целевое состояние, поиск прекращается.
- Любое состояние, имеющееся во множестве закрытых состояний, отбрасывается. Остальные непосещенные состояния добавляются в конец очереди открытых состояний, и поиск продолжается.

Поиск гарантированно находит кратчайший путь к целевому состоянию, если такой путь существует.

Поиск в ширину

Наилучший, средний и наихудший случаи: $O(b^d)$

```
search (initial, goal)
    if initial = goal then return "Решено"
    open = new Queue                                              ❶
    closed = new Set
    insert (open, copy(initial))

    while очередь open не пуста do
        n = head (open)                                           ❷
        insert (closed, n)
        foreach допустимый ход m в позиции n do
            nextState = Состояние, когда из n сделан ход m
            if nextState нет во множестве закрытых состояний then
                if nextState = goal then return "Решено"
                insert (open, nextState)                          ❸

    return "Решения нет"
end
```

❶ Поиск в ширину использует очередь для хранения открытых состояний, которые должны быть посещены.
❷ Удаление самого старого состояния из очереди.
❸ Вставка очередного состояния будет операцией добавления, так как open представляет собой очередь.

Анализ алгоритма поиска в ширину

- Поиск должен хранить во множестве открытых состояний порядка b^d состояний доски, где b - коэффициент ветвления состояний доски, а d — глубина найденного решения (для поиска в глубину этот параметр равен $b\dot{d}$).
- Поиск в ширину гарантированно находит решение с наименьшим числом ходов, которые преобразуют начальное состояние доски в целевое состояние.
- Поиск добавляет состояние доски во множество открытых состояний, только если оно отсутствует во множестве закрытых состояний.

Алгоритм A*Search

A*Search

представляет собой итеративный, упорядоченный поиск, который поддерживает набор открытых состояний доски, предназначенных для исследования в попытке достичь целевого состояния.

На каждой итерации *A*Search* использует функцию оценки $f(n)$ для выбора того состояния доски n из множества открытых состояний, для которого $f(n)$ имеет наименьшее значение.

$$f(n) = g(n) + h(n)$$

где

- $g(n)$ записывает длину кратчайшей последовательности ходов из начального состояния в состояние доски n; это значение вычисляется по мере работы алгоритма;
- $h(n)$ оценивает длину кратчайшей последовательности ходов из состояния n в целевое состояние.

Алгоритм A*Search

- Функция $f(n)$ оценивает длину кратчайшей последовательности ходов от исходного состояния до целевого, проходящей через n.
- A*Search проверяет достижение целевого состояния только тогда, когда состояние доски удаляется из множества открытых состояний (в отличие от поиска в ширину и поиска в глубину, которые выполняют эту проверку тогда, когда генерируются состояния-преемники).
- Это различие гарантирует, что решение представляет собой наименьшее количество ходов от начального состояния доски, если только $h(n)$ никогда не переоценивает расстояние до целевого состояния.
- Низкая оценка $f(n)$ предполагает, что состояние доски п находится близко к конечному целевому состоянию.

Алгоритм A*Search

- Наиболее важным компонентом $f(n)$ является эвристическая оценка, которую вычисляет функция $h(n)$, поскольку значение $g(n)$ может быть вычислено «на лету» путем записи в каждом состоянии доски его удаленности от первоначального состояния.
- Если $h(n)$ не в состоянии точно отделить перспективные состояния доски от бесперспективных, алгоритм A*Search будет выполняться не лучше, чем уже описанный слепой поиск.
- В частности, функция $h(n)$ должна быть приемлемой, т.е. она никогда не должна преувеличивать фактическую минимальную стоимость достижения целевого состояния.
- Если оценка оказывается слишком высокой, алгоритм A*Search может не найти оптимальное решение.

Однако определить приемлемую эффективно вычисляемую функцию $h(n)$ - трудная задача. Существуют многочисленные примеры непригодных $h(n)$ которые, тем не менее, приводят к практичным, хотя и не оптимальным решениям.

Алгоритм A*Search

Наилучший случай: ($Ob{\cdot}d$); средний и наихудший случаи: $O(b^d)$

```
search (initial, goal)
    initial.depth = 0
    open = new PriorityQueue                                    ❶
    closed = new Set
    insert (open, copy(initial))

    while очередь open не пуста do
        n = minimum (open)
        insert (closed, n)
        if n = goal then return "Решено"
        foreach допустимый ход m в состоянии n do
            nextState = состояние после хода m в состоянии n
            if closed содержит состояние nextState then continue

            nextState.depth = n.depth + 1
            prior = состояние в open, соответствующее nextState ❷
            if prior отсутствует
               или nextState.score < prior.score then
                if prior имеется в open                         ❸
                    remove (open, prior)                        ❹
                insert (open, nextState)                        ❺

    return "Решения нет"
end
```

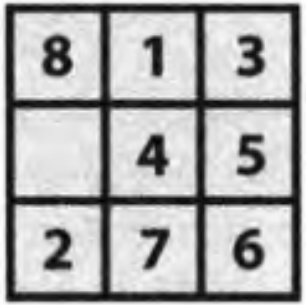
1. A*Search хранит открытые состояния в очереди с приоритетами в соответствии с вычисленной оценкой.
2. Требуется возможность быстро найти соответствующий узел в open.
3. Если A*Search повторно посещает в open состояние prior, которое теперь имеет меньшую оценку...
4. ... то заменяем состояние prior в open альтернативой с лучшей оценкой.
5. Поскольку open является очередью с приоритетами, состояние nextState вставляется в нее с учетом его оценки.

Входные и выходные параметры алгоритма A*Search

- Алгоритм начинает работу с начального состояния доски в дереве поиска и целевого состояния.
- Он предполагает наличие функции оценки $f(n)$ с приемлемой функцией $h(n)$.
- Алгоритм возвращает последовательность ходов, являющуюся решением, которое наиболее близко приближается к решению с минимальной стоимостью, переводящему первоначальное состояние в целевое (или объявляет, что такое решение не может быть найдено при имеющихся в наличии ресурсах).

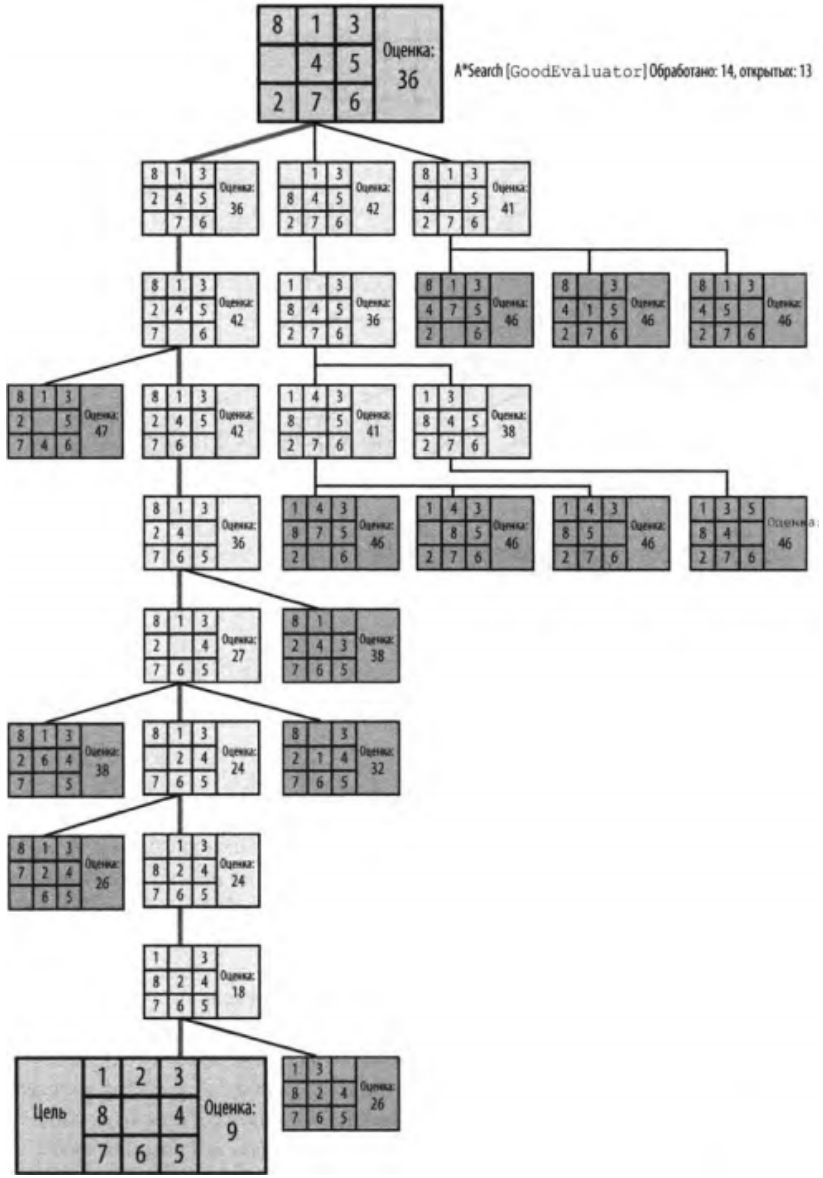
Пример работы алгоритма

Начальное состояние доски для алгоритма A *Search:

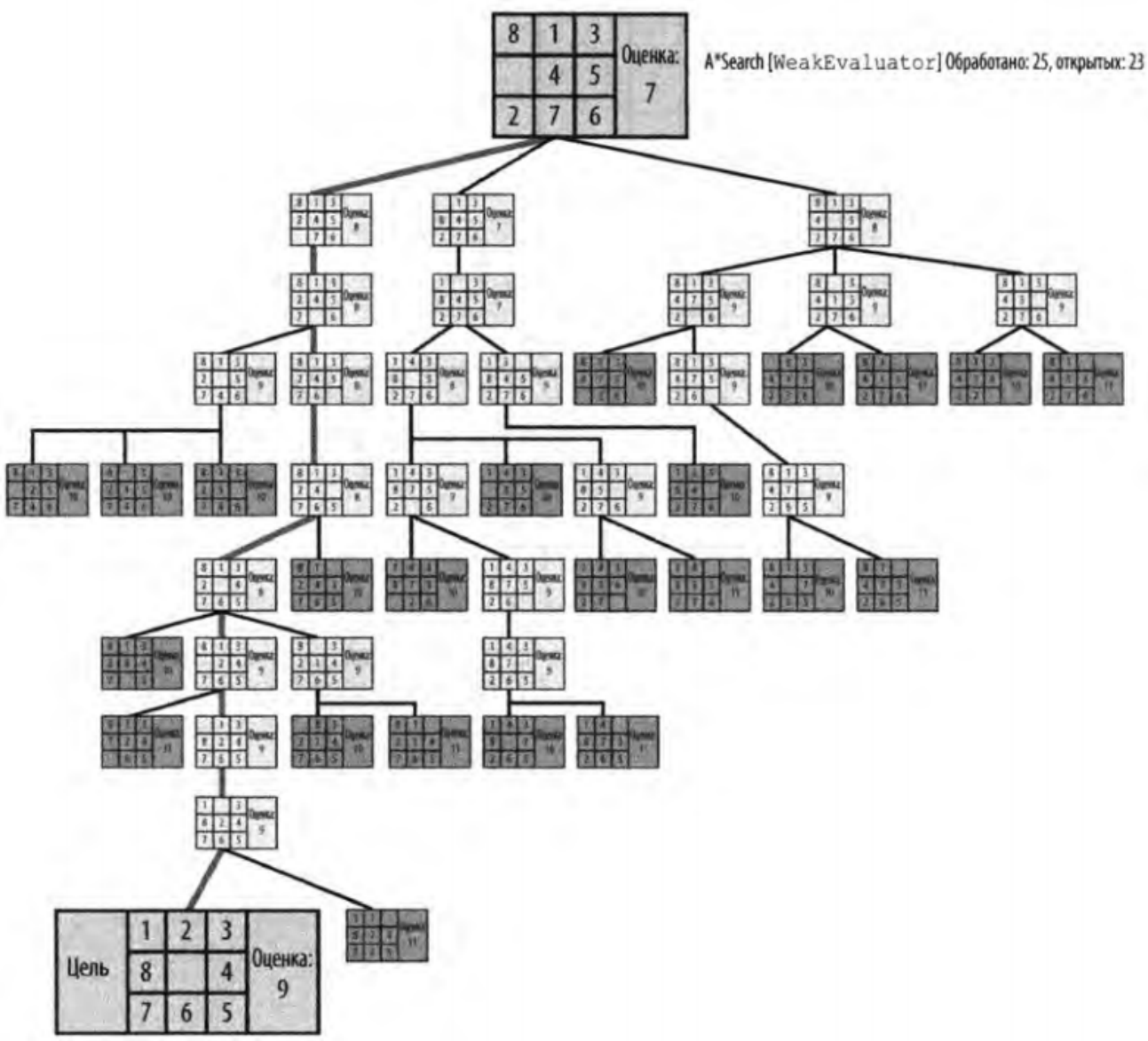


- Для примера головоломки «8» с начальным состоянием, показанным на рисунке.
- Дерево на следующем слайде использует функцию $f(n)$ *GoodEvaluator*, предложенную в *Nilsson, N., Problem-Solving Methods in Artificial Intelligence. McGraw-Hill, 1971.*
- Через слайд использована функция *WeakEvaluator* из той же работы.
- Светло-серый цвет состояния показывает множество открытых состояний в момент достижения целевого состояния.

Пример дерева A*Search при использовании функции GoodEvaluator



Пример дерева A*Search при использовании функции WeakEvaluator



- Обе функции находят одно и то же решение из девяти ходов в целевой узел, но поиск с использованием функции GoodEvaluator оказывается более эффективным.
- Заметим, что через два хода от первоначального состояния в дереве поиска GoodEvaluator начинается четкий путь из узлов с постоянно уменьшающимся значением $f(n)$, который приводит к целевому узлу.
- В дереве же поиска WeakEvaluator приходится исследовать четыре хода от исходного состояния до того, как направление поиска будет сужено.
- WeakEvaluator неверно различает состояния доски; значение $f(n)$ целевого узла оказывается выше, чем значения f(n) для начального узла и всех трех его дочерних узлов.

- Успех алгоритма A*Search непосредственно зависит от его эвристической функции.
- Компонент $h(n)$ функции оценки $f(n)$ должен быть тщательно разработан, и это в большей степени искусство, чем наука.
- Если h(n) всегда равно нулю, A*Search вырождается в простой поиск в ширину.
- Кроме того, если $h(n)$ переоценивает стоимость достижения целевого состояния, A*Search может не суметь найти оптимальное решение, хотя и сможет возвратить какое-то решение в предположении, что $h(n)$ не слишком далека от истины.
- A*Search будет находить оптимальное решение, если эвристическая функция $h(n)$ является приемлемой.

- Большая часть литературы, посвященной A*Search, описывает высокоспециализированные функции $h(n)$ для разных областей, таких как поиск маршрута на цифровой местности [1] или планирование проектов при ограниченных ресурсах [2].
- Книга [3] представляет собой обширный справочник по разработке эффективных эвристик.
- В [4] описывается, как создавать приемлемые функции $h(n)$, а в [5] приведены последние перспективы использования эвристик в решении задач, и не только для A*Search.

1. Wichmann, D. and B. Wuensche, «Automated route finding on digital terrains», Proceedings of IVCNZ, Akaroa, New Zealand, pp. 107-112, November 2004, https://www.researchgate.net/publication/245571114_Automated_Route_Finding_on_Digital_Terrains.
2. Hartmann, S., Project Scheduling Under Limited Resources: Models, Methods, and Applications. Springer, 1999.
3. Pearl, J., Heuristics: Intelligent Search Strategies for Computer Problem Solving. AddisonWesley, 1984
4. Korf, R. E., «Recent progress in the design and analysis of admissible heuristic functions», Proceedings, Abstraction, Reformulation, and Approximation: 4th International Symposium (SARA), Lecture notes in Computer Science #1864: 45-51, 2000, http://www.aaai.org/Papers/AAAI/2000/AAAI00-212.pdf.
5. Michalewicz, Z. and D. Fogel, How to Solve It: Modern Heuristics. Second Edition. Springer, 2004.

Функции h(n) для головоломки «>8>

- FairEvaluator ::= (n), где (n) - это сумма манхэттенских расстояний каждой плитки от ее окончательного местоположения.
- GoodEvaluator ::= $P(h) + 3 - S(h)$, где (n) определена выше, а $S(n)$ представляет собой оценку последовательности, которая по очереди проверяет нецентральные квадраты, давая 0 для каждой плитки, за которой идет ее корректный преемник, и 2 для плитки, не обладающей этим свойством; плитка в центре получает оценку 1.
- WeakEvaluator - Подсчет количества плиток, находящихся не на своем месте.
- BadEvaluator - Общая разность противоположных (относительно центра) плиток по сравнению с идеальным значением, равным 16.

Пример состояния доски для тестирования функций оценки

Сравнение трех приемлемых и одной неприемлемой функций h(n):

Название	**Вычисление *h*(*n*)**	**Статистика**		
		Решение, ходов	**Закрытых состояний**	**Открытых состояний**
GoodEvaluator	13 + 3*11 = 46	13	18	15
FairEvaluator	13	13	28	21
WeakEvaluator	7	13	171	114
BadEvaluator	9	19	1496	767

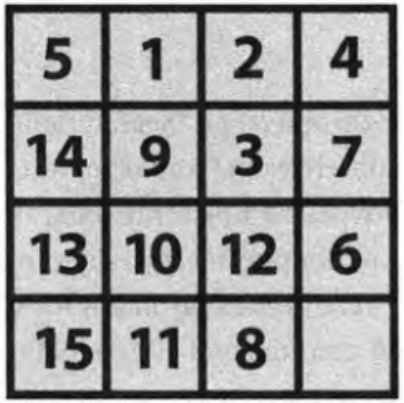
- Чем более сложными становятся состояния доски, тем большую важность приобретают эвристические функции и тем более сложной становится их разработка.
- Они должны оставаться эффективно вычисляемыми, иначе процесс поиска резко замедлится.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

2	10	8	3
1	6		4
5	9	7	11
13	14	15	12

- Для целевого состояния (слева) и начального состояния (справа) A*Search GoodEvaluator быстро находит решение из 15 ходов после обработки 39 состояний доски.
- По окончании работы во множестве открытых состояний остаются 43 непросмотренных состояния.

При попытке решения для более сложного начального состояния алгоритм A*Search исчерпывает всю доступную память.



В таблице содержатся итоговые результаты 1000 испытаний, где n — количество случайных ходов (от 2 до 14):

- данные о среднем количестве состояний в сгенерированных деревьях поиска
- среднее количество ходов в найденных решениях.
- BFS означает поиск в ширину, DFS - поиск в глубину и A - поиск A*Search.

n	#A*	#BFS	#DFS(n)	#DFS(2n)	sA*	sBFS	sDFS(n)	sDFS(2n)
2	4	4,5	3	6,4	2	2	2	2
3	6	13,3	7,1	27,3	3	3	3	3
4	8	25,7	12,4	68,3	4	4	4	5
5	10	46,4	21,1	184,9	5	5	5	5,8
6	11,5	77,6	31,8	321	6	6	6	9,4 (35)
7	13,8	137,9	56,4	767,2	6,8	6,8	6,9	9,7 (307)
8	16,4	216,8	84,7	1096,7	7,7	7,7	7,9 (36)	12,9 (221)
9	21	364,9	144	2520,5	8,7	8,6	8,8 (72)	13,1 (353)
10	24,7	571,6	210,5	3110,9	9,8	9,5	9,8 (249)	16,4 (295)
11	31,2	933,4	296,7	6983,3	10,7	10,4	10,6 (474)	17,4 (364)
12	39,7	1430	452	6196,2	11,7	11,3	11,7 (370)	20,8 (435)
13	52,7	2337,1	544,8	12464,3	13,1	12,2	12,4 (600)	21,6 (334)
14	60,8	3556,4	914,2	14755,7	14,3	13,1	13,4 (621)	25,2 (277)

Темпы роста для слепых поисков приближены следующими функциями:

$$BFS(n) \cong 0,24\dot{(}n+1)^{2.949}$$

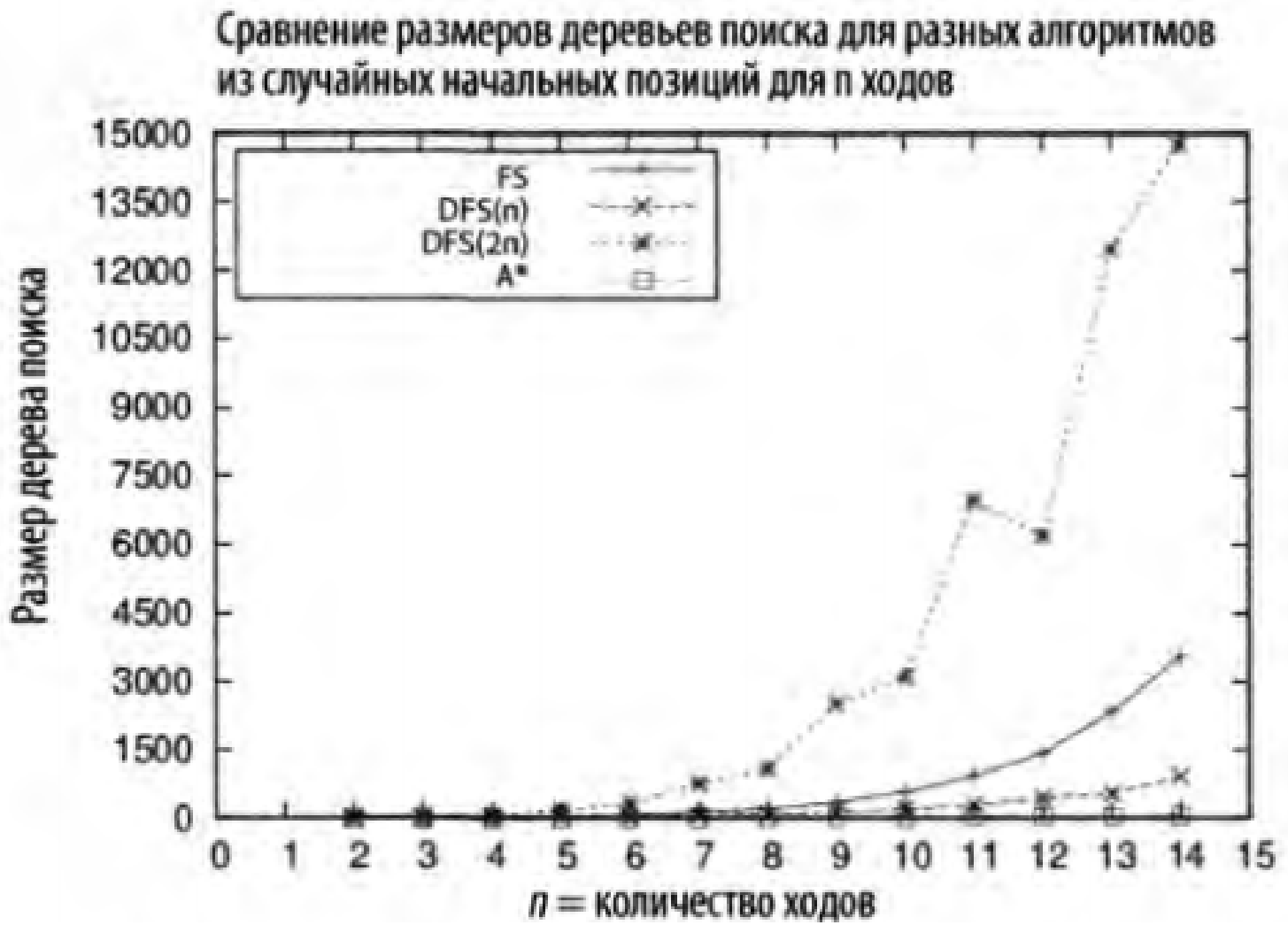
$$DFS(n) \cong 1,43\dot{(}n+1)^{2,275}$$

$$DFS(2n) \cong 3,18\dot{(}n+1)^{3,164}$$

.

В отдельных испытаниях для A*Search с 30 случайными ходами темп роста дерева поиска был оценен как $0(n^{1.5147})$, хотя этот рост и не линейный, размер дерева оказывается значительно меньше, чем для слепых поисков.

Сравнение размеров деревьев поиска для случайных позиций



Сравнение размеров деревьев поиска для случайных позиций