Heuristic Search

Blai Bonet

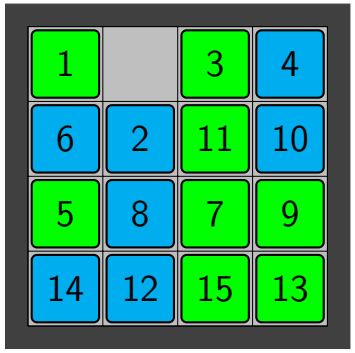
Universidad Simón Bolívar, Caracas, Venezuela

Model for search: the state-space model

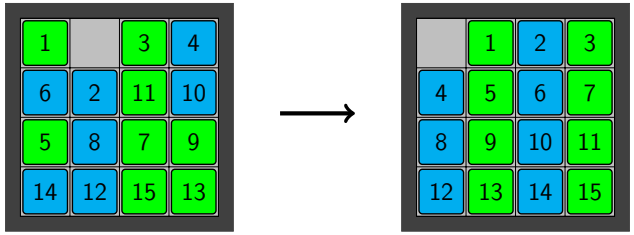
Goals for the lecture

- Understand the state-space model and its applications
- Learn the mathematical formulation of the state-space model
- Learn correspondence between the state-space model and the induced directed graph
- Review basic concepts and facts about graphs

Motivation: the 15-puzzle



Solving the 15-puzzle



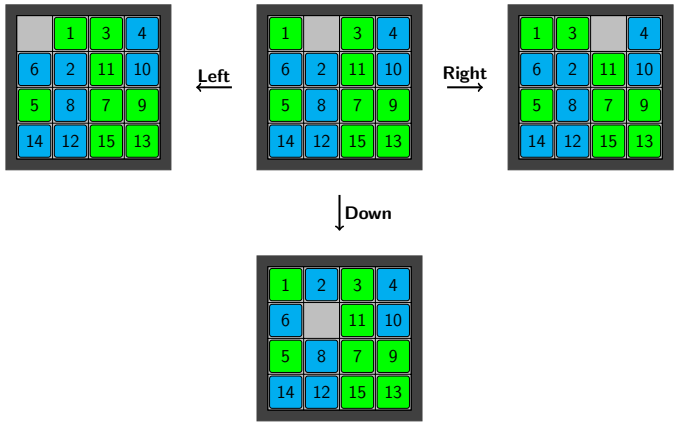
Given the configuration on the left, we want a solution that "transforms" the left into the right configuration

We may want:

– any solution in the form of a sequence of movements

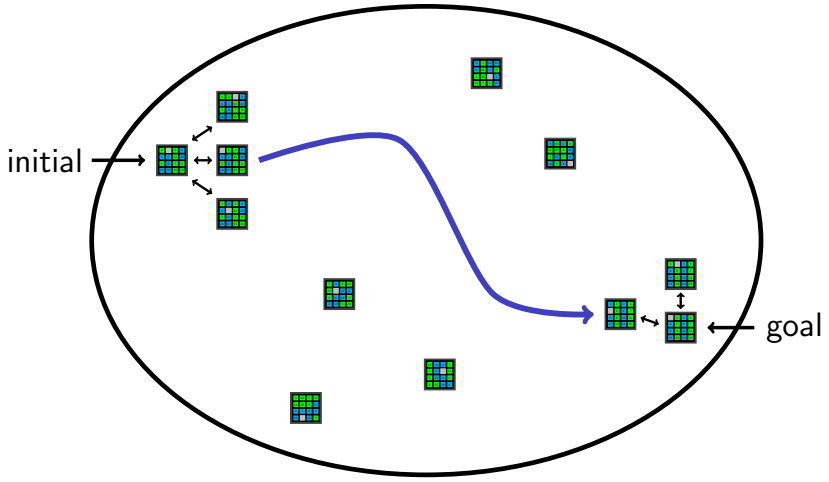
– a solution that is optimal in terms of the number of movements

Movements on the 15-puzzle



The "blank" can move in four directions: left, right, up and down

The 15-puzzle as a directed graph



Space of configurations for the 15-puzzle

There are 16! different configurations of the board that correspond to the different permutations of the 15 tiles plus the blank

However, only 16!/2 configurations are **reachable** from any initial configuration

16!/2 = 10,461,394,944,000 = **10 trillion** plus something states!

If we could use 1 bit of storage/state, we'll need 1.2 Tb of memory!

The state-space model

The state-space model is a mathematical **abstraction** of a problem

An instance of the model is a tuple $P = \langle S, A, s_{init}, S_G, f, c \rangle$ where:

- S is a **finite set** of states
- A is a **finite set** of operators (labels) where $A(s) \subseteq A$ is the subset of operators that can be **applied** at the state s
- s_{init} is the **initial state**
- S_G is a subset of states known as the set of **goal states**
- $f : S \times A \to S$ is a **deterministic transition function** that tells what is the result of applying operator a on state s: $f(s, a)$ is the resulting state
- $c : S \times A \to \mathbb{R}^{\geq 0}$ is a non-negative **cost function** that tells what is the cost of applying operator a on state s: $c(s, a)$ is the cost

The 15-puzzle as an instance of the model

- States $S =$ the 16! permutations of the tiles
- Operators $A = \{\text{Left}, \text{Right}, \text{Up}, \text{Down}\}$
- Applicable operators $A(s)$ determined by the position of blank at s: "blank cannot move outside the board"
- Initial state $s_{init} =$ 
- Goal states $S_G = \left\{ \right.$  $\left. \right\}$; i.e., **unique goal state**
- **Deterministic transitions** that "move the blank" appropriately
- **Unit costs** $c(s, a) = 1$ for all $s \in S$ and $a \in A(s)$

The concept of solution

A solution for a model $P = \langle S, A, s_{init}, S_G, f, c \rangle$ is a sequence of operator (labels) $\sigma = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ that "map" the initial state into a goal state

Formally:

$\sigma = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ is a solution iff it defines a **path** or **trajectory** $\tau = \langle s_0, a_0, s_1, a_1, \ldots, s_n \rangle$ in **state space** such that:

- s_0 is the initial state s_{init}
- a_i is applicable at state s_i for $0 \leq i < n$; i.e. $a_i \in A(s_i)$
- s_i is a valid successor state for $1 \leq i \leq n$; i.e. $s_i = f(s_{i-1}, a_{i-1})$
- s_n is a goal state; i.e. $s_n \in S_G$

Solution costs and optimal solutions

The **cost of solution** $\sigma = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ is the sum of the costs for its operators:

$$c(\sigma) \doteq \sum_{i=0}^{n-1} c(s_i, a_i) = c(s_0, a_0) + c(s_1, a_1) + \cdots + c(s_{n-1}, a_{n-1})$$

where $\langle s_0, a_0, s_1, a_1, \ldots, s_n \rangle$ is the path defined by σ

A solution σ^* is **optimal** if it is a solution of minimum cost; i.e.

$$c(\sigma^*) = \min_{\sigma} c(\sigma)$$

where the minimum is over all solutions σ for the model P

Some applications of the model

The state-space model can be applied to diverse problems in AI and Computer Science

Some examples:

- Optimal rectangle packing (from combinatorial optimization)
- Multiple sequence alignment (from computational biology)
- Proteing folding (from computational biology)
- Synthesis of chemical compounds (from computational chemistry)
- Motion planning (from robotics)
- Classical planning (from automated planning in AI)

Relation to directed graphs

The state-space model is closely related to the model of directed graphs (also called digraphs)

In the rest of the lecture, we revise:

- Labeled directed graphs
- The labeled directed graph induced by a state-space model
- Basic terminology, concepts and results from graph theory

Labeled directed graphs

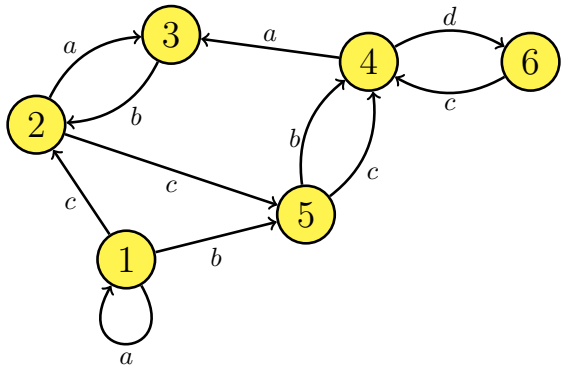
A **labeled** directed graph is a tuple $G = \langle V, L, E \rangle$ where:

- V is set of vertices
- L is set of labels
- $E \subseteq V \times L \times V$ is set of **labeled directed edges**

Element $(v, \ell, v') \in E$ is edge between vertices v and v' with label ℓ

- We assume simple graphs: $(u, \ell, v), (u, \ell, v') \in E \implies v = v'$
- The graph G can be extended with edge costs $c : E \to \mathbb{R}^{\geq 0}$

Labeled directed graph: example



Vertices $V = \{1, 2, 3, 4, 5, 6\}$ Labels $L = \{a, b, c, d\}$

Edges $E =$ "shown in picture"

Induced labeled directed graph

The state model $P = \langle S, A, s_{init}, S_G, f, c \rangle$ induces a labeled directed graph with edge costs $G(P) = \langle V, L, E, c \rangle$ where:

- The vertices V are the states S
- The labels L are the actions A
- The labeled directed edges E are the triplets (s, a, s') such that:
  - s is a state
  - a is an action applicable at s; i.e. $a \in A(S)$
  - s' is the state that results after applying a at s; i.e. $s' = f(s, a)$
- The edge costs $c(s, a, f(s, a))$ are given by the costs $c(s, a)$

Graphs (revision)

Graphs: basic terminology and facts

Let $G = \langle V, E \rangle$ be a graph (it can be undirected, **directed**, labeled)

Consider a vertex $v \in V$ and edge $e \in E$:

- Edge e is **incident** at vertex v if there is x such that $e = (v, x)$ or $e = (x, v)$; in such case, we also say that v is incident at e
- The **degree** of v is $\delta(v) \doteq \#$ of edges incident at v
- The **in-degree** of v is $\delta^-(v) \doteq \#$ of edges of form $(\cdot, v)$ (i.e. number of edges that "enter" v)
- The **out-degree** of v is $\delta^+(v) \doteq \#$ of edges of form $(v, \cdot)$ (i.e. number of edges that "leave" v)

Graphs: basic terminology and facts

Let $G = \langle V, E \rangle$ be a graph (it can be undirected, **directed** or labeled)

- $\delta(v) = \delta^-(v) + \delta^+(v)$
- $\sum_{v \in V} \delta(v) = \sum_{v \in V} \delta^-(v) + \sum_{v \in V} \delta^+(v) = 2|E|$

Graphs: paths and cycles

Let $G = \langle V, E \rangle$ be a graph (it can be undirected, **directed** or labeled)

– A **path** of **lenght** k from vertex u to vertex v is a sequence $\sigma = \langle v_0, v_1, \ldots, v_k \rangle$ of $k+1$ vertices such that $v_0 = u$, $v_k = v$, and $(v_i, v_{i+1}) \in E$ for all $0 \leq i < k$

– The path σ contains the vertices $v_0, v_1, \ldots, v_k$, and the edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$

– σ is a **simple path** if all vertices are different; i.e. $v_i = v_j \Rightarrow i = j$

– σ is a **cycle** if $v_0 = v_k$ and it contains at least one edge

– It is a **simple cycle** if it is a cycle and every **proper prefix** is a simple path; equivalently, $v_i = v_j \iff i = j \ \vee \ \{i, j\} = \{0, k\}$

Graphs: distances

Let $G = \langle V, E \rangle$ be a graph (it can be undirected, **directed** or labeled)

The **distance** $\delta(u, v)$ between the pair of vertices (u, v) is the **length** of a **shortest path** from u to v

If there is no path from u to v, $\delta(u, v)$ is defined as ∞

Remarks:

– $\delta(u, v)$ shouldn't be confused with degree functions $\delta(\cdot)$, $\delta^-(\cdot)$ or $\delta^+(\cdot)$

– $\delta(u, v)$ is always well defined although there may be more than one shortest path from u to v, or no path at all

– Distances $\delta(u, v)$ are graph properties

Graphs: minimum-cost paths

If the graph has a cost function $c : E \to \mathbb{R}$, then we can assign costs $c(\sigma)$ to paths $\sigma = \langle v_0, v_1, \ldots, v_k \rangle$ as

$$c(\sigma) \doteq \sum_{i=0}^{n-1} c(v_i, v_{i+1}) = c(v_0, v_1) + \cdots + c(v_{n-1}, v_n)$$

The **minimum-cost distance** $\delta_c(u, v)$ between the pair of vertices (u, v) is the minimum cost of a path from u to v

If there is no path from u to v, $\delta_c(u, v)$ is defined as ∞

Remarks:

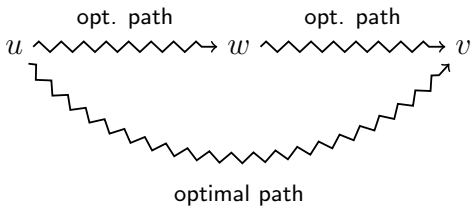
– $\delta(u, v)$ may be $-\infty$ if for each integer α there is a path σ with $c(\sigma) < \alpha$ (can only happen if there is a **cycle of negative cost** "between" u and v)

– When clear from context, we write δ instead of δ_c

Graphs: recursive decomposition of optimal paths

Let $\delta(\cdot, \cdot)$ denote distances or costs of minimum-cost paths in a graph

– For any three vertices $u, v, w \in V$: $\delta(u, v) \leq \delta(u, w) + \delta(w, v)$



Graphs: recursive decomposition of optimal paths

Let $\delta(\cdot,\cdot)$ denote distances or costs of minimum-cost paths in a graph

- For any three vertices $u, v, w \in V$: $\delta(u,v) \leq \delta(u,w) + \delta(w,v)$
- In particular, if (u,w) is an edge in the graph

$$\delta(u,v) \leq \delta(u,w) + \delta(w,v) \leq c(u,w) + \delta(w,v)$$

- If w **belongs** to a minimum-cost path from u to v, the **subpaths** $u \rightsquigarrow w$ and $w \rightsquigarrow v$ are minimum-cost paths, and

$$\delta(u,v) = \delta(u,w) + \delta(w,v)$$

Graphs: undirected vs. directed graphs

An undirected graph is a **special case** of a directed graph

Every undirected edge $u - v$ can be thought as two directed edges $u \rightarrow v$ and $u \leftarrow v$

In a labeled **directed** graph, if there are edges $u \xrightarrow{\ell_1} v$ and $u \xleftarrow{\ell_2} v$, we say that edge ℓ_1 is **invertible** and that edge ℓ_2 is an **inverse** of ℓ_1

In a labeled **undirected** graph, **every edge** is invertible

Summary

- The sliding-tile puzzle known as the 15-puzzle
- Mathematical formulation of state-space model and concepts of solutions and optimal solutions
- The 15-puzzle as an instance of the state-space model
- The induced labeled directed graph
- Basic concepts and facts about graphs

Representation of state spaces

Goals for the lecture

- Learn about the representation problem for state spaces (graphs)
- Introduce three approaches for representation: explicit, black-box (implicit), declarative (implicit)
- Learn about different explicit representations and their space requirements
- Learn about the black-box representation and API
- Learn about high-level PSVN specifications

The representation problem

How do we represent state spaces in practical terms?

How do we cope with the **combinatorial explosion** of states?

Three approaches for representation

We consider three representations of state spaces:

– **Explicit** representation

– Implicit **black-box** representation

– Implicit **declarative** representation

Explicit representation

States have no internal structure; states are **opaque objects**

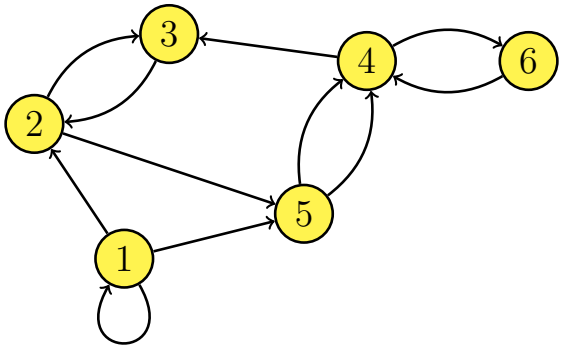
The graph can be represented either by an **adjacency matrix**, by an **incidence matrix**, or by **adjacency lists**

In all cases, the representation is

– **Flat** (i.e., states have no structure)

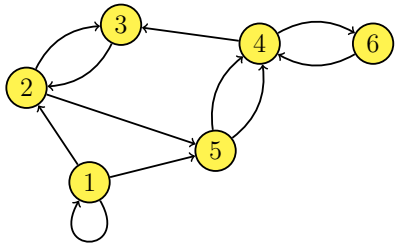
– **Polynomial in size** (in terms of number of vertices and edges)

Explicit representation: example



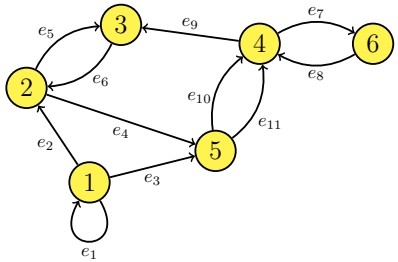
Multi-graph $G = \langle V, E \rangle$ with vertices $V = \{1, 2, 3, 4, 5, 6\}$ and edges
$E = \{(1,1), (1,2), (1,5), (2,3), (2,5), (3,2), (4,3), (4,6), (5,4), (5,4), (6,4)\}$

Explicit representation: adjacency matrix



Adjacency matrix $= \begin{Bmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{Bmatrix}$

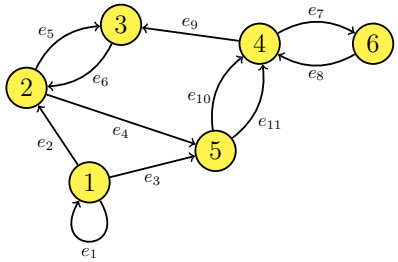
Explicit representation: incidence matrix



Incidence matrix $=$

$$\begin{Bmatrix} 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & -1 & -1 & -1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \end{Bmatrix}$$

Explicit representation: adjacency lists



Adjacency lists $= \begin{cases} \text{edges leaving vertex 1:} & e_1, e_2, e_3 \\ \text{edges leaving vertex 2:} & e_4, e_5 \\ \text{edges leaving vertex 3:} & e_6 \\ \text{edges leaving vertex 4:} & e_7, e_9 \\ \text{edges leaving vertex 5:} & e_{10}, e_{11} \\ \text{edges leaving vertex 6:} & e_8 \end{cases}$

Explicit representation: trade offs

Adjacency matrix:

- space $= O(V^2)$
- time for scan of vertex successors $= O(V)$

Incidence matrix:

- space $= O(VE)$
- time for scan of vertex successors $= O(E)$

Adjacency lists:

- space $= O(V + E)$
- time for scan of vertex successors $= O(\delta(v))$ where $\delta(v)$ is degree of vertex v

All are infeasible for big problems such as the 15-puzzle!

Black-box representation

Structure of the graph is **wired into functions** that provide the necessary information to explore the state space

This information is:

- initial state s_{init}
- goal states S_G
- applicable actions and successor states for given state

A succinct and efficient representation is feasible when states are structures that store information

Black-box representation

Structure of the graph is **wired into functions** that provide the necessary information to explore the state space

This information is:

- `init()`: **provides the initial state**
- `is-goal()`: **tells if state is goal**
- `successors()`: **provides successor states and actions**

A succinct and efficient representation is feasible when states are structures that store information

Black-box representation: API

```
1  typedef unsigned Action
2
3  struct AbstractState {
4      bool is-goal()
5      list<pair<State, Action> > successors()
6  }
7
8  struct State : AbstractState {
9      % ... [internal representation of concrete state]
10
11     bool is-goal()
12     list<pair<State, Action> > successors()
13 }
14
15 State init() % returns initial state
```

Black-box representation: example for 15-puzzle

```
1  struct State15Puzzle : AbstractState {
2      char pos[16]  % pos[i] contains the 'tile' in ith-position
3      char blank    % contains position of blank
4
5      bool is-goal()
6          for i = 0 to 15
7              if pos[i] != i return false
8          return true
9
10     list<pair<State, Action> > successors()
11 }
```

This representation consumes $17 \times 8 = 136$ bits per state which is a lot compared with other representations

Indeed, a state for the 15-puzzle can be stored in 64 bits (16 positions requiring 4 bits each) ...

Black-box representation: space vs. time

Typically, states can be represented in two manners:

- **Space efficient:** compact form that permits to store more states in available memory. However, operations need to 'decode' and 'encode' states and thus consume time
- **Time efficient:** operations are efficient since there is explicit representation, but states consume more space

Best representation depends on the selected algorithm

Black-box representation: example for 15-puzzle

```
1  struct State15Puzzle2 : AbstractState {
2      uint32 tiles[2] % tiles occupying positions (in order)
3
4      bool is-goal()
5          return (tiles[0] == 0x01234567) & (tiles[1] == 0x89ABCDEF)
6
7      list<pair<State, Action> > successors()
8  }
```

Space-efficient representation requiring 64 bits per state (not the best though...)

Space efficiency doesn't directly translate into time performance

Declarative representation

State spaces are described using a **high-level** and **general** representation languages

Representation languages **provide:**

- generality and flexibility
- abstraction that hide low-level implementation

Representation languages **lack:**

- state-of-the-art efficiency on selected problems

Active research topic is to reduce gap in efficiency between black-box and declarative represention

Declarative representation: examples

- **PSVN:** production systems on fixed-length vectors of labels
- **STRIPS:** propositional representation of planning problems
- **SAS$^+$:** multi-valued variable representation of planning problems

What is PSVN?

System designed by Rob Holte et al. at University of Alberta, Canada

[the following is taken from PSVN tutorial by R. Holte at ICAPS-14]

PSVN is:

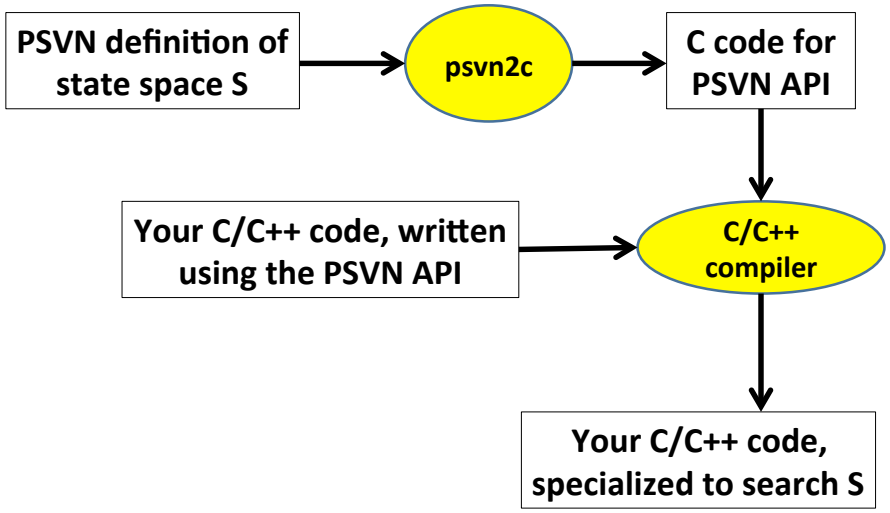
1. **Declarative language** for defining state space models
2. An **API** that includes definitions, constants and functions that implement the state space model
3. A **compiler** that creates C code implementing the API from a **PSVN specification**
4. A **suite of software** implementing other functionalities on top of the PSVN API

PSVN: API

API provides functionality for:

- input/output of states
- testing whether a state is goal
- iterating over successor states
- duplicate pruning based on operator suffixes
- construction, solvability and serialization of abstractions
- etc.

PSVN: workflow



[Image from PSVN tutorial by R. Holte at ICAPS-14]

PSVN: language

Declarative language in which states are defined as **vectors of symbols**

Each vector position contains a symbol from a **fixed finite domain**

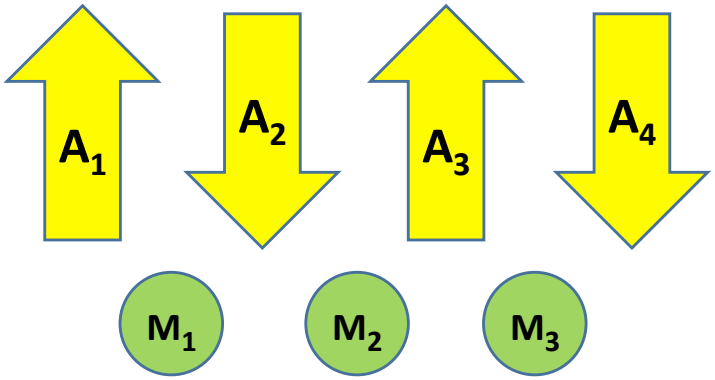
Operators are specified using production-like **transformation rules**

Transformation rules of the form

$$LHS \Rightarrow RHS$$

where LHS specifies the state prior to application of operator and RHS specifies how the application changes the state

PSVN: 4-Arrow puzzle



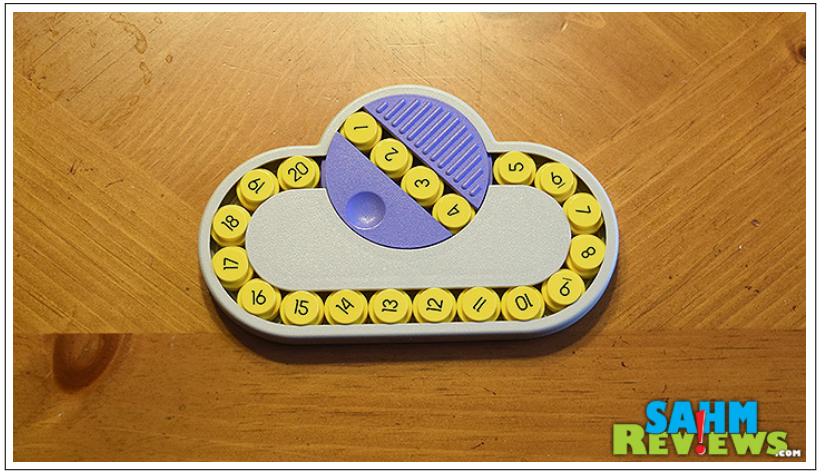
Operator M_i **flips** the arrows A_i and A_{i+1}

[Image from PSVN tutorial by R. Holte at ICAPS-14]

PSVN: rules for M_1 in 4-Arrow puzzle

```
1   # number of state variables
2   4
3
4   # All state variables are binary.
5   2 2 2 2
6
7   0 0 - - => 1 1 - -   LABEL Flip_1_2
8   0 1 - - => 1 0 - -   LABEL Flip_1_2
9   1 0 - - => 0 1 - -   LABEL Flip_1_2
10  1 1 - - => 0 0 - -   LABEL Flip_1_2
11
12  - 0 0 - => - 1 1 -   LABEL Flip_2_3
13  - 0 1 - => - 1 0 -   LABEL Flip_2_3
14  - 1 0 - => - 0 1 -   LABEL Flip_2_3
15  - 1 1 - => - 0 0 -   LABEL Flip_2_3
16
17  - - 0 0 => - - 1 1   LABEL Flip_3_4
18  - - 0 1 => - - 1 0   LABEL Flip_3_4
19  - - 1 0 => - - 0 1   LABEL Flip_3_4
20  - - 1 1 => - - 0 0   LABEL Flip_3_4
21
22  GOAL 0 0 0 0
```


PSVN: Top Spin (video)



[Source http://www.sahmreviews.com/2014/06/thrift-treasure-top-spin.html]

PSVN: 12-4 Top Spin

```
# number of state variables = the number of tokens
12

# All state variables have 12 possible values.
12 12 12 12 12 12 12 12 12 12 12 12

X1 X2 X3 X4 -  -  -  -  -  -  -  -  => X4 X3 X2 X1 -  -  -  -  -  -  -  -  LABEL RevAt0
-  X1 X2 X3 X4 -  -  -  -  -  -  -  => -  X4 X3 X2 X1 -  -  -  -  -  -  -  LABEL RevAt1
-  -  X1 X2 X3 X4 -  -  -  -  -  -  => -  -  X4 X3 X2 X1 -  -  -  -  -  -  LABEL RevAt2
-  -  -  X1 X2 X3 X4 -  -  -  -  -  => -  -  -  X4 X3 X2 X1 -  -  -  -  -  LABEL RevAt3
-  -  -  -  X1 X2 X3 X4 -  -  -  -  => -  -  -  -  X4 X3 X2 X1 -  -  -  -  LABEL RevAt4
-  -  -  -  -  X1 X2 X3 X4 -  -  -  => -  -  -  -  -  X4 X3 X2 X1 -  -  -  LABEL RevAt5
-  -  -  -  -  -  X1 X2 X3 X4 -  -  => -  -  -  -  -  -  X4 X3 X2 X1 -  -  LABEL RevAt6
-  -  -  -  -  -  -  X1 X2 X3 X4 -  => -  -  -  -  -  -  -  X4 X3 X2 X1 -  LABEL RevAt7
-  -  -  -  -  -  -  -  X1 X2 X3 X4 => -  -  -  -  -  -  -  -  X4 X3 X2 X1 LABEL RevAt8
X4 -  -  -  -  -  -  -  -  X1 X2 X3 => X1 -  -  -  -  -  -  -  -  X4 X3 X2 LABEL RevAt9
X3 X4 -  -  -  -  -  -  -  -  X1 X2 => X2 X1 -  -  -  -  -  -  -  -  X4 X3 LABEL RevAt10
X2 X3 X4 -  -  -  -  -  -  -  -  X1 => X3 X2 X1 -  -  -  -  -  -  -  -  X4 LABEL RevAt11

GOAL 0 1 2 3 4 5 6 7 8 9 10 11
GOAL 1 2 3 4 5 6 7 8 9 10 11 0
GOAL 2 3 4 5 6 7 8 9 10 11 0 1
GOAL 3 4 5 6 7 8 9 10 11 0 1 2
GOAL 4 5 6 7 8 9 10 11 0 1 2 3
GOAL 5 6 7 8 9 10 11 0 1 2 3 4
GOAL 6 7 8 9 10 11 0 1 2 3 4 5
GOAL 7 8 9 10 11 0 1 2 3 4 5 6
GOAL 8 9 10 11 0 1 2 3 4 5 6 7
GOAL 9 10 11 0 1 2 3 4 5 6 7 8
GOAL 10 11 0 1 2 3 4 5 6 7 8 9
GOAL 11 0 1 2 3 4 5 6 7 8 9 10
```


PSVN: 12-4 Top Spin (compact encoding)

```
# number of state variables = the number of tokens
12

# All state variables have 12 possible values.
12 12 12 12 12 12 12 12 12 12 12 12

X1 X2 X3 X4 -  -  -  -  -  -  -  -  => X4 X3 X2 X1 -  -  -  -  -  -  -  -  LABEL RevAt0
-  X1 X2 X3 X4 -  -  -  -  -  -  -  => -  X4 X3 X2 X1 -  -  -  -  -  -  -  LABEL RevAt1
-  -  X1 X2 X3 X4 -  -  -  -  -  -  => -  -  X4 X3 X2 X1 -  -  -  -  -  -  LABEL RevAt2
-  -  -  X1 X2 X3 X4 -  -  -  -  -  => -  -  -  X4 X3 X2 X1 -  -  -  -  -  LABEL RevAt3
-  -  -  -  X1 X2 X3 X4 -  -  -  -  => -  -  -  -  X4 X3 X2 X1 -  -  -  -  LABEL RevAt4
-  -  -  -  -  X1 X2 X3 X4 -  -  -  => -  -  -  -  -  X4 X3 X2 X1 -  -  -  LABEL RevAt5
-  -  -  -  -  -  X1 X2 X3 X4 -  -  => -  -  -  -  -  -  X4 X3 X2 X1 -  -  LABEL RevAt6
-  -  -  -  -  -  -  X1 X2 X3 X4 -  => -  -  -  -  -  -  -  X4 X3 X2 X1 -  LABEL RevAt7
T1 T2 T3 T4 T5 T6 T7 T8 X1 X2 X3 X4 => X3 X2 X1 T1 T2 T3 T4 T5 T6 T7 T8 X4 LABEL RevAt8
X4 T1 T2 T3 T4 T5 T6 T7 T8 X1 X2 X3 => X2 X1 T1 T2 T3 T4 T5 T6 T7 T8 X4 X3 LABEL RevAt9
X3 X4 T1 T2 T3 T4 T5 T6 T7 T8 X1 X2 => X1 T1 T2 T3 T4 T5 T6 T7 T8 X4 X3 X2 LABEL RevAt10
X2 X3 X4 T1 T2 T3 T4 T5 T6 T7 T8 X1 => T1 T2 T3 T4 T5 T6 T7 T8 X4 X3 X2 X1 LABEL RevAt11

GOAL 0 1 2 3 4 5 6 7 8 9 10 11
```


Summary

- Approaches for representation: explicit, black-box, declarative
- Explicit representation not generally feasible for state-space models
- Black-box reperesentation is efficient and implemented directly into programming language
- Required API for black-box representation
- PSVN system for fast implementation of APIs
- New problems: n-Arrow and (n, k)-Top Spin

State spaces: case studies

Goals for the lecture

- Illustrate how to formulate a problem described in natural language as an instance of the state-space model
- Travelling in Romania
- Travelling Salesman Problem (TSP)
- Blocksworld
- Tower of Hanoi

Romania

[Image from `http://www.geoatlas.com/medias/maps/countries/romania/ro111a36/romania_phy.jpg`]

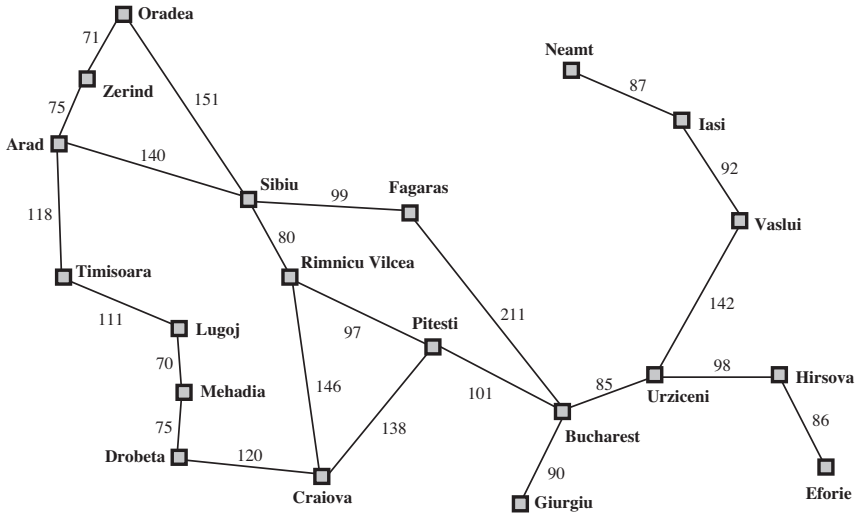
Travelling in Romania [AIMA]

Given two cities in Romania, we want to find the **shortest route** to go from one city to the other

Problem is a particular instance of more general problem in which one wants to navigate a map in an efficient way:

- GPS
- travelling plans
- optimization of delivery routes

Travelling in Romania: simplified routing graph



Travelling in Romania: state space model

Given source and destination cities c_{src} and c_{dst}, the problem of finding a shortes route from c_{src} to c_{dst} corresponds to the instance $P = \langle S, A, s_{init}, S_G, f, c \rangle$ of the state space model where:

- **states** S are the cities of Romania (in simplified routing graph)
- **actions** are of the form $Go(c_i, c_j)$ = "go from city c_i to city c_j"
- **applicable actions** at city c_i are all actions $Go(c_i, c_j)$ such that c_i is connected to c_j in the simplified routing graph
- $s_{init} = c_{src}$
- $S_G = \{c_{dst}\}$
- **transition function** is $f(c_i, Go(c_i, c_j)) = c_j$ for all cities c_i and actions $Go(c_i, c_j) \in A(c_i)$
- **action costs** are given by the numbers attached to the edges in the simplified routing graph; e.g. $c(\text{Lugoj}, Go(\text{Lugoj}, \text{Timisoara})) = 111$

Travelling in Romania: additional information

Straight-line distances to Bucharest

city	sl-distance	city	sl-distance
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Travelling Salesman Problem

Travelling Salesman Problem (TSP)

TSP is an **optimization problem**

TSP is one of the most important problems in CS and OR, and also one of the most studied problems

The TSP is abound finding a **tour of optimal cost** that visits a given set of cities

TSP: formal definition

An instance of the TSP consists of:

- collection $C = \{1, 2, \ldots, n\}$ of n cities each one denoted by an integer in $\{1, 2, \ldots, n\}$
- an $n \times n$ square cost matrix $D = (d_{ij})_{ij}$ such that the cost to **go directly** from city i to city c_j is d_{ij}

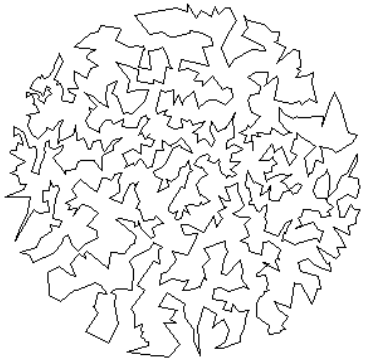
  (If the city c_i is not directly connected with city c_j, $d_{ij} = \infty$)

A tour is a **simple cycle** $\langle c_1, c_2, \ldots, c_n \rangle$ over the n cities.
Equivalently, a tour is a permutation of $\{1, 2, \ldots, n\}$

The cost of a tour $\langle c_1, c_2, \ldots, c_n \rangle$ is $\sum_{i=1}^{n-1} d_{c_i, c_{i+1}} + d_{c_n, c_1}$

An **optimal tour** is a tour of **minimum cost**

TSP: example



[Images from http://www.cs.princeton.edu/courses/archive/fall04/cos126/assignments/tsp.html]

Variants of the TSP

- **Metric TSP:** the distances in the cost matrix satisfy the triangle inequality; i.e. $d_{ij} \leq d_{ik} + d_{kj}$ for all $i, j, k \in \{0, 1, \ldots, n\}$
- **Euclidean TSP:** instance given by a set of n points on the plane. The distance between points is the **euclidean distance** between the points

  Special case of Metric TSP since distances on the plane satisfy the triangle inequality

  (The previous example is an euclidean TSP)
- **Symmetric TSP:** the cost matrix is symmetric
- **Asymmetric TSP:** the cost matrix is not symmetric

Applications of the TSP

- vehicle routing
- circuit board drilling
- VLSI design
- robot control
- X-ray crystallography
- machine scheduling
- computational biology

TSP: state space model

Different formulations of TSP as instance of the state space model

We consider a **naive formulation** in which a tour from a **fixed initial city** (say city 1) is **grown** until becoming a complete tour

The states are **partial tours** starting at 1 while the goal states are complete tours

Later on, we'll consider a better formulation

TSP: naive state space model

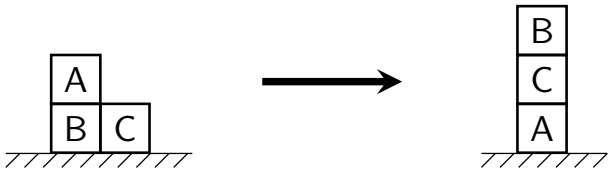
Instances of the form $P = \langle S, A, s_{init}, S_G, f, c \rangle$ where:

- **states** in S are **simple paths** over $C \cup \{\perp\}$ starting at 1 where $\perp$ is a new symbol that marks the end of a tour
- **actions** are $Go(i, j)$ for extending the current path with edge (i, j) and $Finish(i)$ that finishes a partial tour ending at i

  Applicable action $A(s)$ at state s **ending at city** i is $\{Finish(i)\}$ if $|s| = n$, or $\{Go(i, j) : j \notin s\}$ if $|s| < n$
- $s_{init} = \langle 1 \rangle$
- $S_G = \{s \in S : s \text{ ends at } \perp\}$ (i.e. all complete tours)
- **transitions:** $f(s, Go(i, j)) = \langle s, j \rangle$ for states s with $Go(i, j) \in A(s)$, and $f(s, Finish(i)) = \langle s, \perp \rangle$ for states s with $A(s) = \{Finish(i)\}$
- **action costs** $c(s, Go(i, j)) = d_{ij}$ and $c(s, Finish(i)) = d_{i1}$

Blocksworld

Blocksworld



initial state goal state

Actions:

- Move(x, y, z) moves block x that is on top of block y onto top of block z. Blocks x and z must be "clear" (no block on top)
- Unstack(x, y) moves block x that is on y onto table. Block x must be clear. The table always has space for any block
- Stack(x, y) moves block x from table onto top of block y. Blocks x and y must be clear

Blocksworld: state space model

Instance for problem with set $\mathcal{B}$ of blocks is $P = \langle S, A, s_{init}, S_G, f, c \rangle$:

- **States** are all possible configuration of blocks in $\mathcal{B}$ on table
- **Actions:** Move(x, y, z), Unstack(x) and Stack(x, y) for $x, y, z \in \mathcal{B}$
  - Move(x, y, z) is applicable at s iff blocks x and y are clear, and x is on y
  - Unstack(x, y) is applicable at s iff block x is clear and on y
  - Stack(x, y) is applicable at s iff blocks x and y are clear, and x is on table
- $s_{init} =$ initial configuration of blocks
- $S_G =$ set of target configurations of blocks
- **transitions:** "intended" effect of actions in blocksworld
- **action costs:** all actions have unit costs

Tower of Hanoi

Tower of Hanoi

[Image from `http://www.activityresources.com/Tower-of-Hanoi.html`]

Tower of Hanoi

General instance consists of n disks and k pegs

The disks can be in any peg with the sole constraint that no disk can be on top of smaller disk

Problem consists of moving the n disks from one peg (typically 1) to another peg (typically 2) with single-disk movements that do not violate above constraint

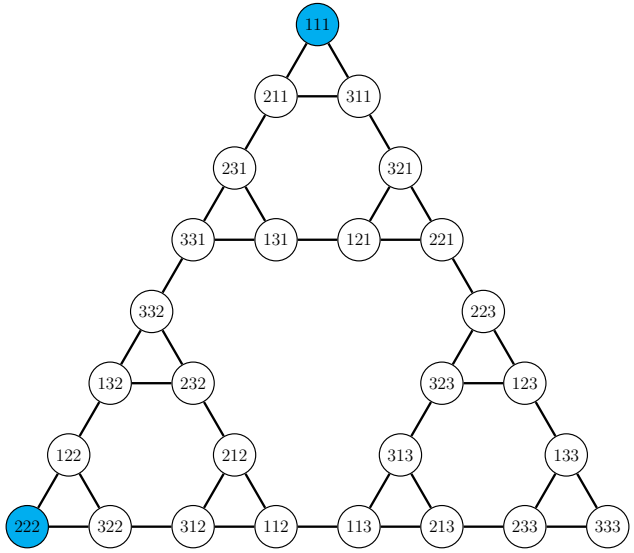
Tower of Hanoi: state space model

Instance for n disks and k pegs is $P = \langle S, A, s_{init}, S_G, f, c \rangle$:

- **States** are all configuration of disks on pegs that satisfy constraint
- **Actions:** Move(d, p, q) for moving the disk d from peg p to peg q

  Move(d, p, q) is applicable at s iff d is on peg p, no disk is on top of d, and peg q has no disks or smallest disk on q is smaller than d
- s_{init} = configuration with all disks on peg 1
- $S_G = \{$configuration with all disks on peg 2$\}$
- **transitions:** "intended" effect of move actions
- **action costs:** all actions have unit costs

Tower of Hanoi: graph for 3 disks and 3 pegs



Tower of Hanoi with 3 pegs

The exact solution for n disks and 3 pegs is **well known**

The solution decomposes **recursively** into three subtasks:

Subtask 1. move $n-1$ disks from peg 1 to peg 3

Subtask 2. move the largest disk from peg 1 to peg 2

Subtask 3. move $n-1$ disks from peg 3 to peg 2

Subtasks 1 and 3 are instances for $n-1$ disks with 3 pegs

If $T(n, k)$ denotes the (minimum) number of moves for solving n disks with k pegs:

$$T(n, 3) \;=\; 2T(n-1, 3) + 1$$

whose solution is $T(n, 3) = 2^n - 1$

Tower of Hanoi with 4 or more pegs

The problem with 4 or more pegs is different

We **don't know** a recursive decomposition nor a closed form for the minimum number of movements to solve it

Obviously $T(n, k) \leq T(n, 3)$ since the algorithm that solves the problem with 3 pegs also solves the problem with k pegs ($k \geq 3$)

The **Frame-Stewart conjecture** claims that for each n and k, there is ℓ such that

$$T(n, k) = 2T(\ell, k) + T(n - \ell, k - 1)$$

The conjecture had been **verified** for 4 pegs up to 30 disks using **heuristic search** [Korf & Felner, IJCAI 2007, 2324–2329]

Summary

- Several problems can be formulated as instances of the state space model

Search trees and basic concepts

Goals for the lecture

- Learn what is the search tree associated to a graph
- Learn how the search tree is stored in memory
- Basic concepts of generation and expansion of nodes, and branching factor
- Illustrate search tree and branching factor in the 15-puzzle
- Present canonical search trees as tool for analysis

Exploring a graph: search trees

Every search algorithm generates a tree when exploring a graph. Such a tree is called a **search tree**

The shape of the tree depends on the algorithm and the order in which the children of nodes are explored

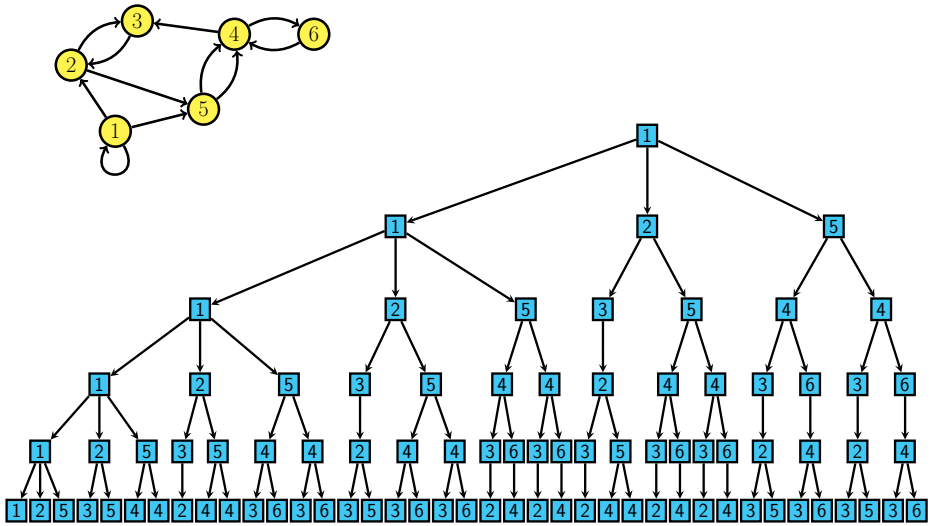
The search algorithm can either:

– store the **complete tree** in memory,

– store only the **current branch** of the search, or

– store partial information about the tree

Search tree: example



Representation of search trees: nodes

Search trees are made of **nodes** that represent states

As in all trees, a node only needs to store a pointer to its **unique parent** (the root is the only node that has no parent)

We also store in nodes:

– the label of the edge connecting the parent with the node

– the cost of the **unique path** from the root to the node

The nodes in the search tree represent states and paths

Node: data structure and basic manipulation

```
struct Node {
    AbstractState state   % pointer to state represented by node
    Node          parent  % parent of node (null if root)
    Action        action  % action mapping state to node (-1 if root)
    unsigned      g       % cost of path: root -> node

    Node(AbstractState s, Node p, Action a, unsigned gcost)
      : state(s), parent(p), action(a), g(gcost)

    Node make-node(AbstractState state, Action action)
        return new Node(state, this, action, g + c(this, action))

    extract-path(vector<Action> &reversed-path)
        Node node = this
        while node != null && node.parent != null
            reversed-path.push-back(node.action)
            node = node.parent
}

Node make-root-node(AbstractState state)
    return new Node(state, null, -1, 0)
```


Generation and expansion of nodes

A **node is generated** when a data structure for it is created in memory

A **node is expanded** when all its children are generated

These are **critical operations** that are performed thousands/millions of times

Average number of nodes generated per second is measure of efficiency

15-puzzle: complete search tree

depth	#nodes	emp. branching factor
0	1	2.00000000
1	2	3.00000000
2	6	3.00000000
3	18	3.22222222
4	58	3.20689655
5	186	3.23655913
6	602	3.23255813
7	1,946	3.23638232
8	6,298	3.23563035
9	20,378	3.23613701
10	65,946	3.23601128
11	213,402	3.23608026
12	690,586	3.23606038
13	2,234,778	3.23606998
14	7,231,898	3.23606693
15	23,402,906	3.23606828
16	75,733,402	3.23606783
17	245,078,426	3.23606802
18	793,090,458	3.23606795
19	2,566,494,618	3.23606798
20	8,305,351,066	3.23606797

15-puzzle: formal analysis of branching factor

It can be shown that the number d_n of nodes at depth n in the **complete search tree** for the 15-puzzle is

$$d_n = \frac{1}{5}\left[(\phi_1 + 2)^2 \times \phi_1^{n-2} + (\phi_2 + 2)^2 \times \phi_2^{n-2} + 2\right]$$

where $\phi_1 = 1 + \sqrt{5} \approx 3.23$ and $\phi_2 = 1 - \sqrt{5} \approx -1.23$

The **asymptotic branching factor**, if it exists, is $\lim_{n\to\infty} d_{n+1}/d_n$

For the complete search tree in the 15-puzzle, the limit exists:

$$\lim_{n\to\infty} \frac{d_{n+1}}{d_n} = \phi_1 = 1 + \sqrt{5} \approx 3.23606797$$

At depth 27, number of nodes in search tree exceeds the total number of states in the problem!

Foundation for analysis: canonical search trees

In order to analyse general search algorithms, we need to assume a nice and regular structure for search trees

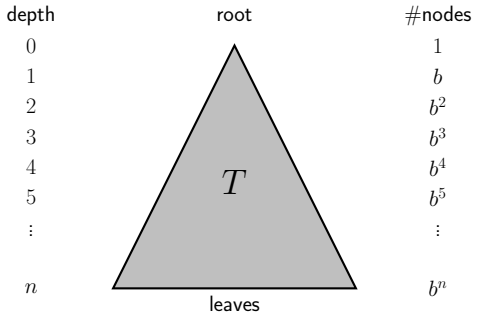
Otherwise the analysis is **too complex** (e.g. see analysis of 15-puzzle)

A **canonical search tree** T satisfies:

- T is a **full tree** (that means that all leaves appear at same depth)
- T has a **regular branching factor** b across all internal nodes

Recall: branching factor of node n is its number of children

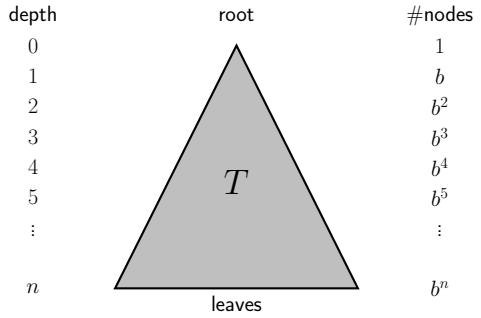
Canonical search tree: number of nodes



Canonical search tree T of depth n and branching factor b:

- Number of nodes at depth d is b^d, and number of leaves is b^n
- Total number of nodes is $N(b,n) \doteq \sum_{k=0}^{n} b^k = \frac{b^{n+1}-1}{b-1} = O(b^n)$
- Ratio $N(b,n)/b^n = \frac{b^{n+1}}{b^n(b-1)} - \frac{1}{b^n(b-1)} < \frac{b^n}{b^n}\frac{b}{b-1} = \frac{b}{b-1}$

Canonical search tree: number of nodes

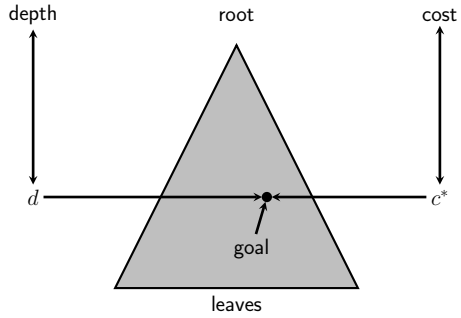


Canonical search tree T of depth n and branching factor b:

– Ratio $N(b,n)/b^n = \frac{b^{n+1}}{b^n(b-1)} - \frac{1}{b^n(b-1)} < \frac{b^n}{b^n}\frac{b}{b-1} = \frac{b}{b-1}$

b	2	3	4	5	6	7
$b/(b-1)$	2.00	1.50	1.33	1.25	1.20	1.16

Canonical search tree: goal depth and cost



Canonical search tree T of depth n and branching factor b:

– Shallowest optimal goal at depth d with cost c^*

– If all costs are equal to 1, then $d = c^*$

– If all costs are positive $\geq c_{\min} > 0$, then $d \leq c^*/c_{\min}$

Summary

- Complete search tree associated with a graph
- Data structure and basic operations for nodes (search tree)
- Concepts: generation, expansion, branching factor
- Complete search tree for the 15-puzzle
- Canonical search trees: full tree of given depth with constant branching factor

Tree search and graph search

Goals for the lecture

- Learn about the problem of duplicates
- Learn what is **tree search** and **graph search**
- Learn how to implement graph search
- Learn about partial removal of duplicates

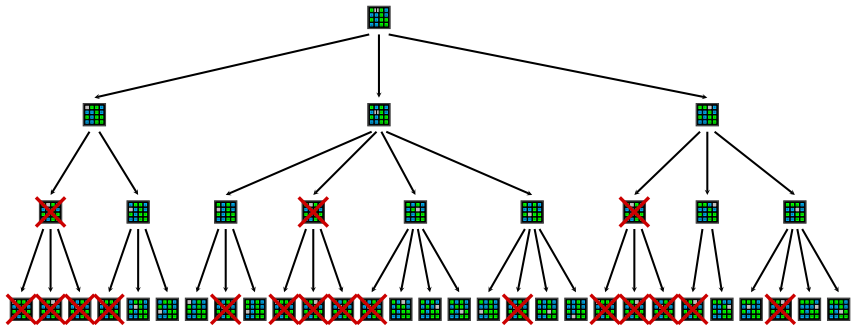
Duplicate nodes

A **duplicate node** in a search tree is a node that is associated with a state that is also referred by other node in the tree

The complete search tree for the 15-puzzle contains many duplicates

Indeed, the number of nodes at depth 27 in the complete search tree is bigger than the total number of states in the problem (i.e. $16!/2$)

15-puzzle: duplicates in complete search tree



At depth 3, more than half of the nodes are duplicates!

Dealing with duplicates

Ideally, we would like to **modify** the search tree so that it contains **no duplicates** and **preserve the shortest paths** in the graph

The best way to remove duplicates is to do it **dynamically** during search by **pruning** the found duplicates

Algorithms that store search trees can do this

Other algorithms may only remove some duplicates

Graph search vs. tree search

A search algorithm that traverses a search tree without duplicates is called a **graph-search algorithm**

Otherwise, it is called a **tree-search algorithm**

A tree-search algorithm may:

– Prune no duplicates (i.e. traverses the complete search tree)

– Prune some duplicates

Graph search

Implementing a graph-search algorithm is **challenging** and requires:

– Every time that a node n is generated, check whether the state associated to n is also associated to another node n'

– In the affirmative, keep the node that refers to the shorter path from the initial state, and discard the other node

The removal of duplicates involve **operations** on the repository of states and the **open list** (the frontier of the explored tree)

15-puzzle: complete search tree

depth	#nodes	emp. branching factor
0	1	2.00000000
1	2	3.00000000
2	6	3.00000000
3	18	3.22222222
4	58	3.20689655
5	186	3.23655913
6	602	3.23255813
7	1,946	3.23638232
8	6,298	3.23563035
9	20,378	3.23613701
10	65,946	3.23601128
11	213,402	3.23608026
12	690,586	3.23606038
13	2,234,778	3.23606998
14	7,231,898	3.23606693
15	23,402,906	3.23606828
16	75,733,402	3.23606783
17	245,078,426	3.23606802
18	793,090,458	3.23606795
19	2,566,494,618	3.23606798
20	8,305,351,066	3.23606797

15-puzzle: graph search

depth	#nodes	depth	#nodes	depth	#nodes	depth	#nodes
0	1	21	3,098,270	42	115,516,106,664	63	105,730,020,222
1	2	22	5,802,411	43	156,935,291,234	64	65,450,375,310
2	4	23	10,783,780	44	208,207,973,510	65	37,942,606,582
3	10	24	19,826,318	45	269,527,755,972	66	20,696,691,144
4	24	25	36,142,146	46	340,163,141,928	67	10,460,286,822
5	54	26	65,135,623	47	418,170,132,006	68	4,961,671,731
6	107	27	116,238,056	48	500,252,508,256	69	2,144,789,574
7	212	28	204,900,019	49	581,813,416,256	70	868,923,831
8	446	29	357,071,928	50	657,076,739,307	71	311,901,840
9	946	30	613,926,161	51	719,872,287,190	72	104,859,366
10	1,948	31	1,042,022,040	52	763,865,196,269	73	29,592,634
11	3,938	32	1,742,855,397	53	784,195,801,886	74	7,766,947
12	7,808	33	2,873,077,198	54	777,302,007,562	75	1,508,596
13	15,544	34	4,660,800,459	55	742,946,121,222	76	272,198
14	30,821	35	7,439,530,828	56	683,025,093,505	77	26,638
15	60,842	36	11,668,443,776	57	603,043,436,904	78	3,406
16	119,000	37	17,976,412,262	58	509,897,148,964	79	70
17	231,844	38	27,171,347,953	59	412,039,723,036	80	17
18	447,342	39	40,271,406,380	60	317,373,604,363		
19	859,744	40	58,469,060,820	61	232,306,415,924		
20	1,637,383	41	83,099,401,368	62	161,303,043,901		

Data from [Korf & Schultze, AAAI-2005]

Partial pruning of duplicates

The idea is to use **limited amount of memory** to prune some duplicates from the search tree

The most frequent techniques for partial pruning are:

– **Pruning parents and grandparents** (for invertible edges)

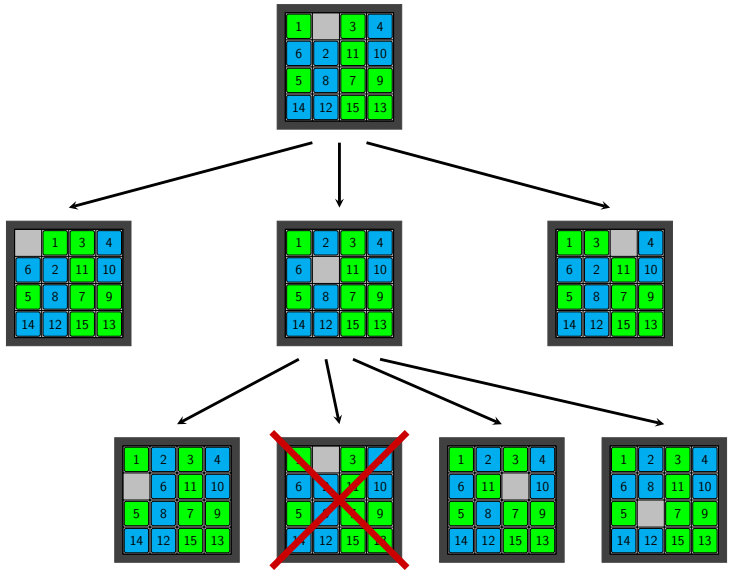
– Use **automata** to prune branches

– Use **(transposition) tables** to store a limited amount of nodes

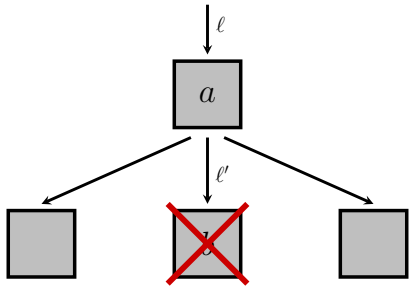
Very often only a **constant amount of memory** is dedicated to pruning

Techniques used in problems with constant costs but it can be adapted for non-constant costs

Simplest pruning: pruning parents



Pruning parents



Node b is a **duplicate** (of a's parent) if operator ℓ' is inverse of ℓ

In 15-puzzle, the operators Left $\leftrightarrow$ Right and Up $\leftrightarrow$ Down are inverses of each other

15-puzzle: search tree with removal of parents

depth	#nodes	emp. branching factor
0	1	2.00000000
1	2	3.00000000
2	6	3.00000000
3	18	3.22222222
4	58	3.20689655
5	186	3.23655913
6	602	3.23255813
7	1,946	3.23638232
8	6,298	3.23563035
9	20,378	3.23613701
10	65,946	3.23601128
11	213,402	3.23608026
12	690,586	3.23606038
13	2,234,778	3.23606998
14	7,231,898	3.23606693
15	23,402,906	3.23606828
16	75,733,402	3.23606783
17	245,078,426	3.23606802
18	793,090,458	3.23606795
19	2,566,494,618	3.23606798
20	8,305,351,066	3.23606797

search tree without removal of parents

depth	#nodes	emp. branching factor
0	1	2.00000000
1	2	2.00000000
2	4	2.50000000
3	10	2.40000000
4	24	2.25000000
5	54	2.00000000
6	108	2.01851851
7	218	2.16513761
8	472	2.18220338
9	1,030	2.13980582
10	2,204	2.11705989
11	4,666	2.11744534
12	9,880	2.13016194
13	21,046	2.13684310
14	44,972	2.13310504
15	95,930	2.12880225
16	204,216	2.12859913
17	434,694	2.13018813
18	925,980	2.13108058
19	1,973,338	2.13083415
20	4,204,856	2.13028888

search tree with removal of parents

Asymptotic branching factor ≈ 2.13 (compare with 3.23)

Pruning with automata: duplicate-generating suffix

Removal of parents is a special case of a more general technique

Consider a path $\sigma = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ in the search tree and the decomposition $\sigma = \sigma_0\omega$ where ω is the suffix $\omega = \langle a_{k+1}, \ldots, a_{n-1} \rangle$ for some $k < n$

We say that ω **generates duplicates relative to** σ_0 when the node $n(\sigma)$ for the path σ is equal to the node $n(\sigma')$ for the path $\sigma' = \sigma_0\omega'$ for some $\omega' = \langle a'_{k+1}, \ldots . a'_{k+j} \rangle$ **shorter than** ω (i.e. $k + j < n - 1$)

For example, in the 15-puzzle, the suffix $\omega = \langle \text{Right}, \text{Left} \rangle$ generates duplicates relative to **any prefix** σ_0

Pruning with automata: invalid suffixes

Suppose that the suffix ω generates a duplicate relative to σ_0

We can prune all branches that contain the path $\sigma_0\omega$ since all nodes in such branches are duplicates of higher cost

For example, in the 15-puzzle, we can prune all branches containing $\langle \text{Right}, \text{Left} \rangle$

We say that a suffix ω is an **invalid suffix** if ω generates duplicates relative to any prefix σ_0

Pruning with automata

We can use a set L of invalid suffixes to prune duplicates

Whenever we generate a new node $n = n(\sigma)$ associated with path σ, prune n if some suffix of σ belongs to L

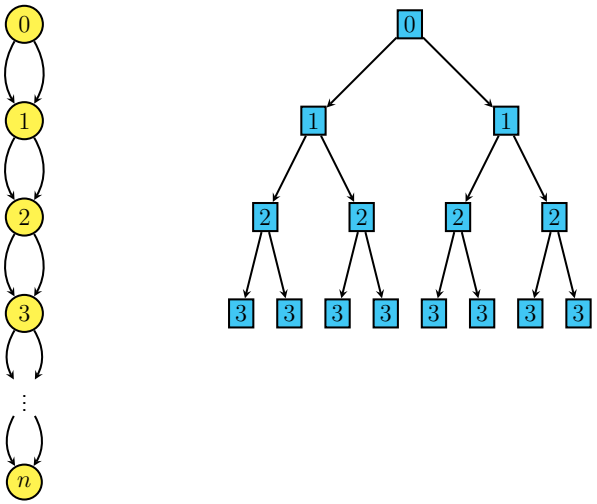
However, checking whether some suffix of σ belongs to L can be a relative expensive operation

Better is to compute an **automata** A that recognizes the **language** L^R and then run σ^R over A to see if σ contains a suffix in L

In practice,

- one constructs a set L of invalid suffixes up to certain length ℓ
- pruning power of L does not increase at constant rate with ℓ (diminishing returns)

A graph with an exponential search tree



Summary

- Duplicates: what is a duplicate, dealing with duplicates
- Graph search vs. tree search
- Partial pruning of duplicates
- Existence of exponential number of duplicates

Breadth-first search and uniform-cost search

Goals for the lecture

- Learn breadth-first search and uniform-cost search
- Tree-search and graph-search variants of the algorithms
- Fundamental characteristics
- Analysis on canonical search trees

Breadth-first search

Breadth-first search explores the search tree **layer by layer** expanding all nodes at depth d before expanding any node at depth $d + 1$

Nodes get ordered for expansion using a **FIFO queue**

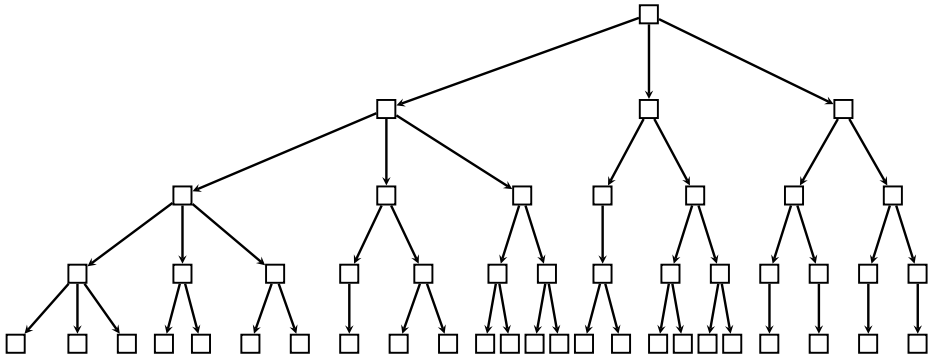
Algorithm can be implemented as a tree- or graph-search algorithm depending on whether **duplicates** are pruned or not

Breadth-first search is a fundamental algorithms in CS

Breadth-first search for explicit graphs [CLRS]

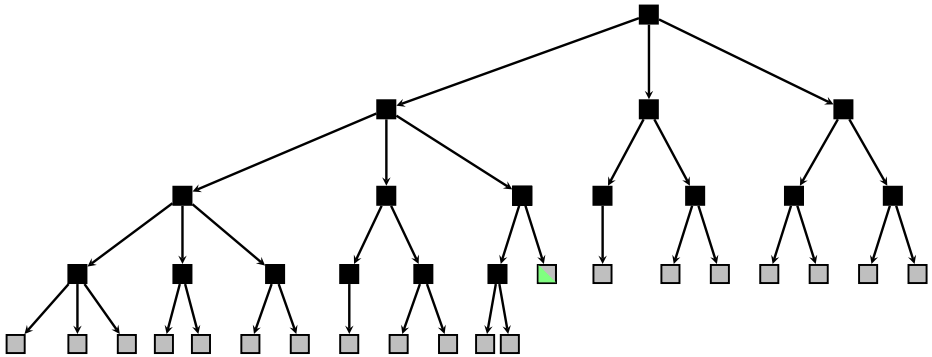
```
1  breadth-first-search(Vertex root):
2
3      % initialization
4      foreach Vertex u
5          color[u] = White
6          distance[u] = ∞
7          parent[u] = null
8
9      Queue q                  % FIFO queue
10     color[root] = Gray
11     distance[root] = 0
12     q.insert(root)
13
14     % search
15     while !q.empty()
16         Vertex u = q.pop()
17         foreach Vertex v in adj[u]
18             if color[v] == White
19                 color[v] = Gray
20                 distance[v] = distance[u] + 1
21                 parent[v] = u
22                 q.insert(v)
23         color[u] = Black
```


Breadth-first search: example



search tree

Breadth-first search: example



first in queue

Breadth-first search for implicit graphs

```
1  % breadth-first search without duplicate elimination
2  breadth-first-search():
3
4      % initialization
5      Queue q
6      Node root = make-root-node(init())
7      q.insert(root)
8
9      % search
10     while !q.empty()
11         Node n = q.pop()
12
13         % check for goal
14         if n.state.is-goal() return n
15
16         % expand node
17         foreach <s,a> in n.state.successors()
18             q.insert(n.make-node(s, a))
19
20     return null        % failure: there is no path from root to goal
```

Recall API and basic primitives

```
struct AbstractState {
    bool is-goal()
    list<pair<State, Action> > successors()
}

State init()

struct Node {
    Node(AbstractState s, Node p, Action a, unsigned gcost)
      : state(s), parent(p), action(a), g(gcost)

    Node make-node(AbstractState state, Action action)
        return new Node(state, this, action, g + c(this, action))

    extract-path(vector<Action> &reversed-path)
        Node node = this
        while node != null && node.parent != null
            reversed-path.push-back(node.action)
            node = node.parent
}

Node make-root-node(AbstractState state)
    return new Node(state, null, -1, 0)
```


Differences between breadth-first searches

CLRS's breadth-first search is for explicit graphs, ours is for implicit graphs and thus explores the search tree associated to a graph

Our breadth-first search returns as soon as a node associated to a goal state is selected for expansion (why not generated?)

Breadth-first search for implicit graphs

```
% breadth-first search with duplicate elimination
breadth-first-search():
    % initialization
    Queue q
    set-color(init(), Gray)
    q.insert(make-root-node(init()))

    % search
    while !q.empty()
        Node n = q.pop()

        % check for goal
        if n.state.is-goal() return n

        % expand node
        foreach <s,a> in n.state.successors()
            if get-color(s) == White
                set-color(s, Gray)
                q.insert(n.make-node(s, a))
        set-color(n.state, Black)

    return null        % failure: there is no path from root to goal
```


Extended API for duplicate elimination I

```

void set-color(AbstractState state, Color color)

unsigned get-color(AbstractState state)

```

Later on we'll see how to implement these functions efficiently

Properties of breadth-first search

Expands all nodes at depth d before expanding any at depth $d+1$

- **Complete:** if there is a path, outputs a path, else outputs `null` (if duplicates pruned)
- **Optimality:** it returns a **shortest** path (not min cost!)
- **Time complexity:** $O(b^d)$ (i.e. exponential in goal depth d)
- **Space complexity:** $O(b^d)$ (i.e. exponential in goal depth d)

Time and space complexities calculated in **canonical search tree** with branching factor b where **shallowest** goal appears at depth d

Uniform-cost search

Breadth-first search finds shortest paths rather than **minimum-cost** paths from the initial state to a goal state

Uniform-cost search explores search tree **cost-layer by cost-layer** expanding all nodes at cost c before expanding any node at cost $> c$

Nodes get ordered for expansion using a **priority queue**

Like breadth-first search, algorithm can be implemented as tree- or graph-search depending on whether duplicates are pruned or not

Uniform-cost search for implicit graphs

```
1  % uniform-cost search with duplicate elimination
2  uniform-cost-search():
3      % initialization
4      PriorityQueue q                    % ordered by node's g
5      set-color(init(), Gray)
6      set-distance(init(), 0)
7      q.insert(make-root-node(init()), 0)
8
9      % search
10     while !q.empty()
11         Node n = q.pop()
12
13         % check for goal
14         if n.state.is-goal() return n
15
16         % expand node
17         expansion-for-uniform-cost-search(n, q)
18         set-color(n.state, Black)
19
20     return null        % failure: there is no path from root to goal
```


Uniform-cost search: expansion

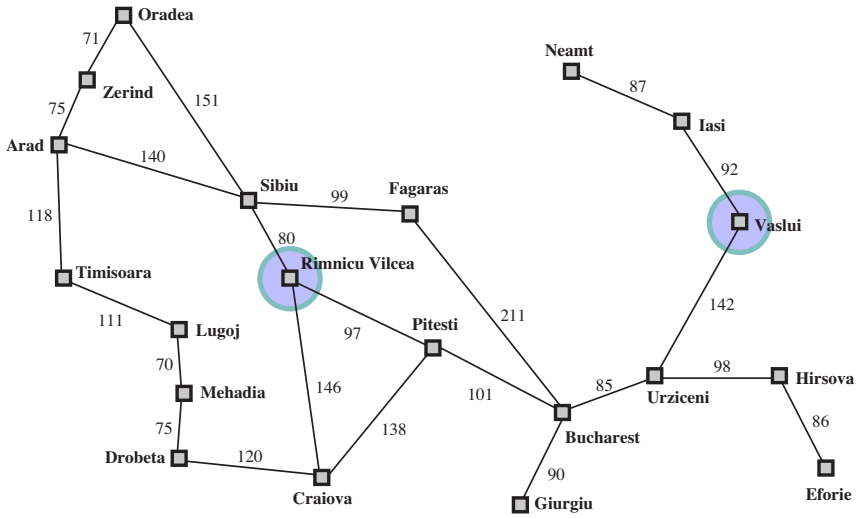
```
21 expansion-for-uniform-cost-search(Node n, PriorityQueue q):
22     foreach <s,a> in n.state.successors()
23         g = n.g + c(n.state, a)
24
25         % is it the first time we see the state?
26         if get-color(s) == White
27             set-color(s, Gray)
28             set-distance(s, g)
29             q.insert(n.make-node(s,a), g)
30
31         % a shorter path was found, update open list
32         else if g < get-distance(s)
33             assert(get-color(s) == Gray)       % why?
34             set-distance(s, g)
35             s.node.parent = n
36             s.node.action = a
37             s.node.g = g
38             q.decrease-priority(s.node, g)
```

Extended API for duplicate elimination II

```
1
2  void set-color(AbstractState state, Color color)
3
4  unsigned get-color(AbstractState state)
5
6  void set-distance(AbstractState state, unsigned dist)
7
8  unsigned get-distance(AbstractState state)
9
```

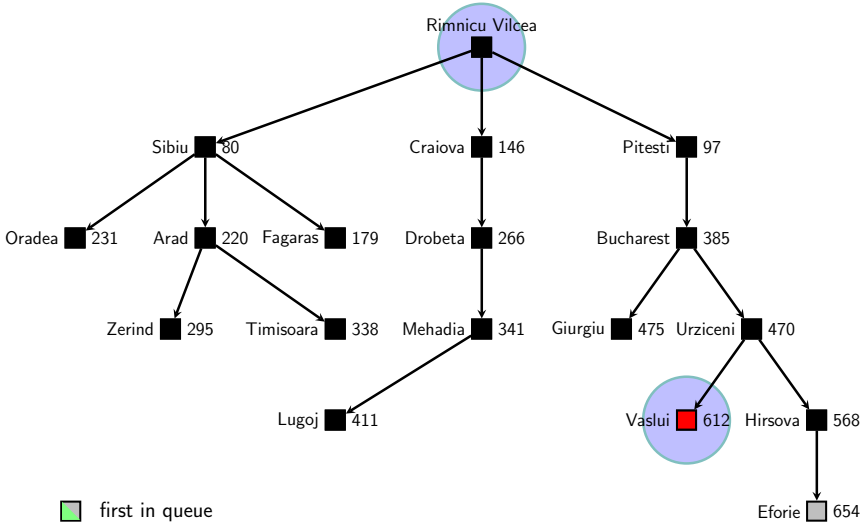
Later on we'll see how to implement these functions efficiently

Traveling in Romania [AIMA]



Task: find optimal path from **Rimnicu Vilcea** to **Vaslui**

Uniform-cost search: example



Delayed duplicate elimination

Weaker form of duplicate elimination in which duplicates are allowed to exist in the open list (nodes generated that are waiting for expansion)

We call this **delayed duplicate elimination**

It can be thought as that the search explicates **graph edges** rather than **graph vertices**

UCS with delayed duplicate elimination

```
1  % uniform-cost search with delayed duplicate elimination
2  uniform-cost-search():
3      PriorityQueue q                    % ordered by node's g
4      set-color(init(),Gray)
5      set-distance(init(), 0)
6      q.insert(make-root-node(init()), 0)
7
8      % search
9      while !q.empty()
10         Node n = q.pop()
11         State ns = n.state
12
13         % delayed duplicate elimination
14         if n.g < get-distance(ns)
15             set-distance(ns, n.g)
16             if ns.is-goal() return n
17
18             % expand node
19             foreach <s,a> in ns.successors()
20                 set-color(s, Gray)
21                 set-distance(s, n.g+c(ns,a))
22                 q.insert(n.make-node(s,a), n.g+c(ns,a))
23             set-color(ns,Black)
24     return null        % failure: there is no path from root to goal
```

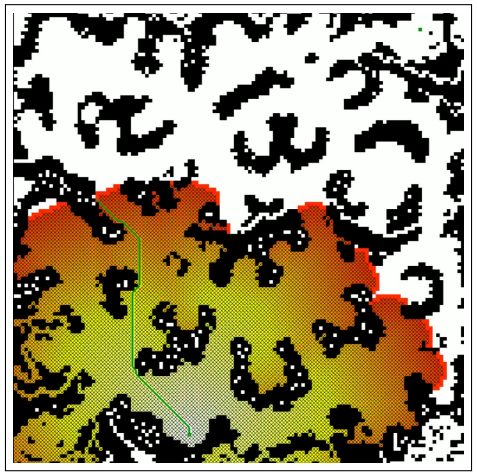

Properties of uniform-cost search

- **Complete:** if there is a path, outputs a path else outputs `null` (if duplicates pruned)
- **Optimality:** it returns a **minimum-cost** path
- **Time complexity:** $O(b^{c^*/c_{\min}})$ $(= O(b^d)$ if all costs are 1)
- **Space complexity:** $O(b^{c^*/c_{\min}})$ $(= O(b^d)$ if all costs are 1)

If duplicates aren't pruned and minimum edge cost $c_{\min} = 0$, UCS may not terminate even if there is a path from init to goal

Time and space complexities calculated in **canonical search tree** with branching factor b and $c_{\min} > 0$ where **minimum-cost** goal appears at depth d with cost c^*

Uniform-cost search on a pathfinding problem (video)



[Video by Jordan Thayer. `Http://www.jordanthayer.com`]

Summary

- Breadth-first search
- Properties of breadth-first search
- Uniform-cost search
- Properties of uniform-cost search
- Delayed duplicate detection (search explicates edges rather than nodes)

Depth-first search and iterative deepening depth-first search

Goals for the lecture

- How to deal with time and space
- Depth-first search: incomplete and suboptimal, but linear space
- Iterative depth-first search: incomplete*, optimal and linear space

Dealing with time and space

Time: just wait: go for a coffee, go for lunch, go on a trip, ...

Space: buy more memory. Expensive ... may work in some cases ...

It's easier and cheaper to manage time than space!

Depth-first search

Depth-first search explores the search tree in **depth-first** manner always expanding the most recent generated node

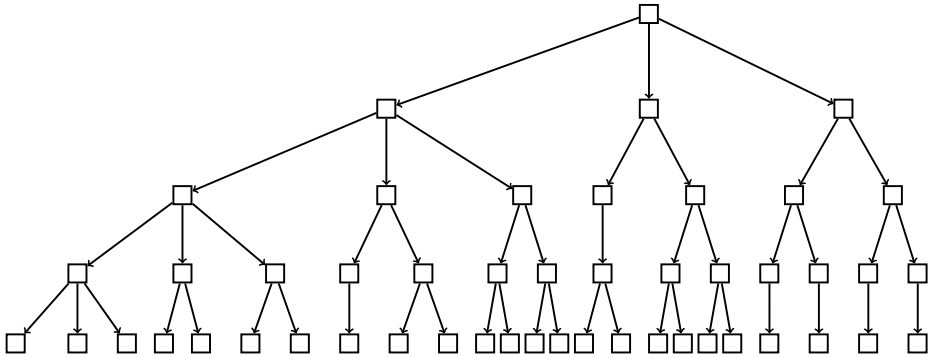
Nodes can be ordered for expansion with a **LIFO queue**, but it is easier to formulate the search **recursively**

Depth-first search is typically implemented as a tree-search algorithm (i.e. minimal duplicate elimination)

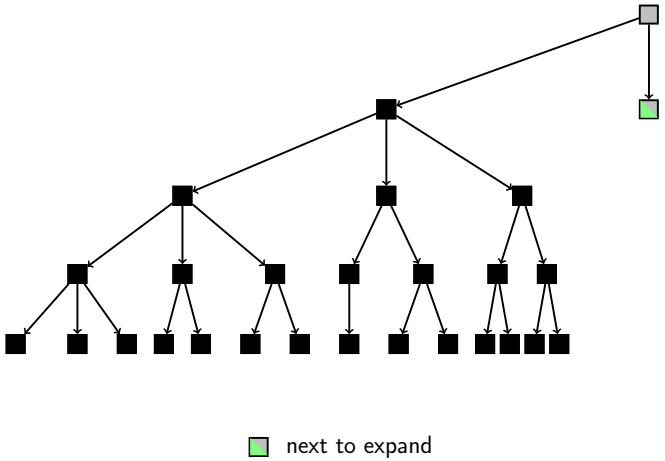
Depth-first search for explicit graphs [CLRS]

```
1  void depth-first-search(Vertex root):
2
3      % initialization
4      foreach Vertex u
5          color[u] = White
6          distance[u] = ∞
7          parent[u] = null
8
9      % depth-first visit on root
10     dfs-visit(root)
11
12
13 void dfs-visit(Vertex u):
14     color[u] = Gray          % time of discovery of vertex u
15     foreach Vertex v in adj[u]
16         if color[v] == White
17             parent[v] = u
18             distance[v] = distance[u] + 1
19             dfs-visit(v)
20     color[u] = Black         % time of finalisation of vertex u
```


Depth-first search: example



Depth-first search: example



Depth-first search for implicit graphs

```
1  % depth-first search without duplicate elimination
2  Node depth-first-search():
3      return dfs-visit(make-root-node(init()))
4
5
6  Node dfs-visit(Node n):
7      if n.state.is-goal() return n
8
9      foreach <s,a> in n.state.successors()
10         Node m = dfs-visit(n.make-node(s, a))
11         if m != null return m
12
13     return null       % failure: there is no path from node n to goal
```

Advantages and disadvantages of depth-first search

Advantages:

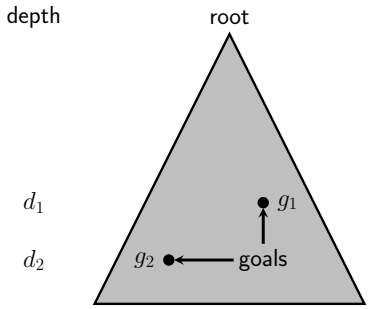
- Easy to implement
- Efficient in space: only need to store current branch and sibling nodes along branch
- **Space requirement is linear** in depth of graph (for branching bounded by constant)

Disadvantages:

- Incomplete (if tree is infinite, it can get stuck in an infinite branch)
- Suboptimal: it may return a suboptimal path to goal

Challenge: develop an optimal and linear-space algorithm

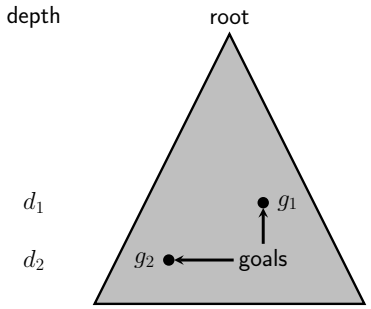
Motivation for iterative deepening depth-first search



Consider above search tree with two goal nodes: an optimal one (g_1) at depth d_1 and another (g_2) at depth $d_2 > d_1$

Consider a depth-first search algorithm that **only dives** up to depth d

Motivation for iterative deepening depth-first search

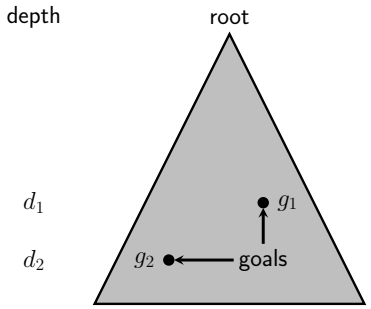


Applying the algorithm at the root of the tree, we obtain:

- if $d < d_1$, the search does not find any goal
- if $d_1 \leq d < d_2$, the search finds the optimal goal g_1
- if $d_2 \leq d$, the search finds the suboptimal goal g_2

In all cases, the algorithm uses **space that is linear** in the depth d

Motivation for iterative deepening depth-first search



Therefore, if we knew the **optimal depth** d, we could find an optimal path to the goal in linear space

Problem: we don't know the optimal depth ... but we can search it!

Iterative deepening depth-first search

Iterative deepening depth-first search does:

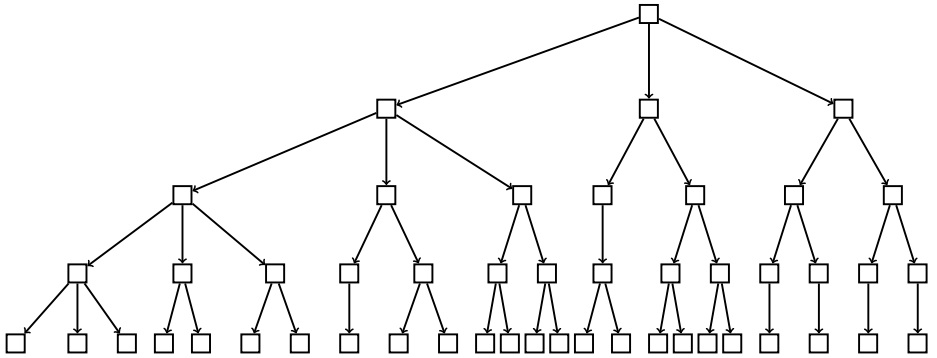
- repeated **depth-bounded** depth-first searches
- with **increasing bounds** of $0, 1, 2, 3, \ldots$ for each search
- until reaching a goal node

It is almost equivalent to breadth-first search but uses less memory:

- it visit the nodes within each iteration in a depth-first order
- but the nodes in the tree are **discovered** in a breadth-first order

It is a **linear space** algorithm and a **tree-search algorithm**
(i.e. it performs partial pruning)

Iterative deepening depth-first search: example



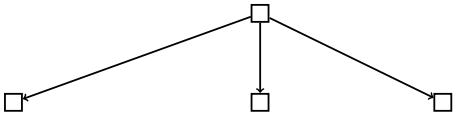
Iterative deepening depth-first search: example

Subtree explored by the bounded depth-first search:

First iteration: depth-first search with bound 0

Iterative deepening depth-first search: example

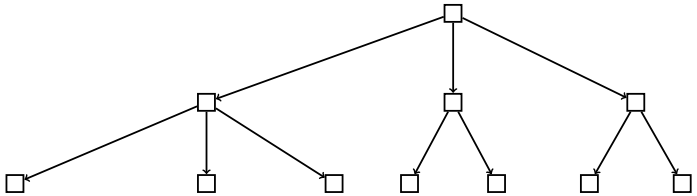
Subtree explored by the bounded depth-first search:



Second iteration: depth-first search with bound 1

Iterative deepening depth-first search: example

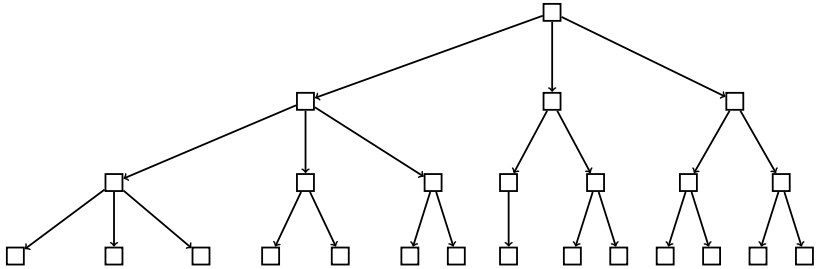
Subtree explored by the bounded depth-first search:



Third iteration: depth-first search with bound 2

Iterative deepening depth-first search: example

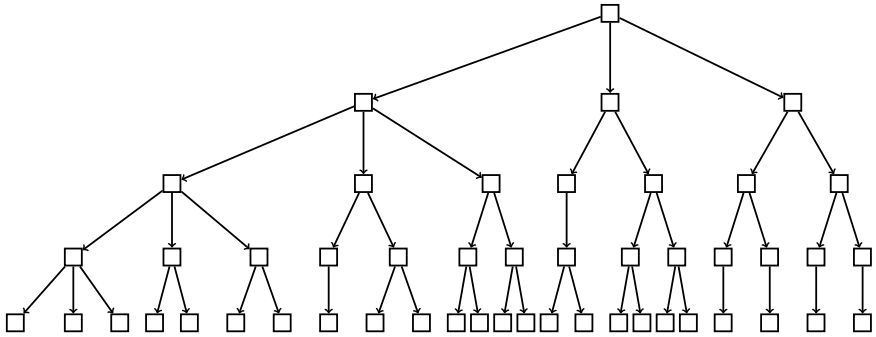
Subtree explored by the bounded depth-first search:



Fourth iteration: depth-first search with bound 3

Iterative deepening depth-first search: example

Subtree explored by the bounded depth-first search:



Fifth iteration: depth-first search with bound 4

Iterative deepening depth-first search: pseudocode

```
1  Node iterative-deepening-depth-first-search():
2      Node root = make-root-node(init())
3      bound = 0
4
5      % perform depth-bounded searches with increasing depth bounds
6      while true
7          Node n = bounded-dfs-visit(root, 0, bound)
8          if n != null return n
9          bound = bound + 1
10
11
12 Node bounded-dfs-visit(Node n, unsigned d, unsigned bound):
13     % base cases
14     if d > bound return null          % prune branch by depth bound
15     if n.state.is-goal() return n     % goal is found, return
16
17     % expansion and recursion
18     foreach <s,a> in n.state.successors()
19         Node m = bounded-dfs-visit(n.make-node(s,a), d+1, bound)
20         if m != null return m
21     return null         % failure: there is no path from node n to goal
```

Analysis

Iterative deepening depth-first search does:

- Complete depth-first searches for trees of depth $0, 1, 2, \ldots, d$
- The root is **generated** $d+1$ times, its children d times, its grandchildren $d-1$ times, etc.

total # of generated nodes across all searches

$$\begin{aligned} &\leq (d+1) + db + (d-1)b^2 + \cdots + b^d \\ &= \sum_{k=0}^{d}(d+1-k)b^k = \sum_{k=1}^{d+1} kb^{d+1-k} \\ &= b^{d+1}\sum_{k=1}^{d} kb^{-k} \leq b^{d+1}\sum_{k=1}^{\infty} kb^{-k} = b^{d+1}\frac{b}{(b-1)^2} = O(b^d) \end{aligned}$$

for constant b

Properties of iterative deepening depth-first search

- **Partially Complete:** if goal is reachable, outputs a path (else iterative deepening depth-first search **doesn't terminate** if search tree is infinite)
- **Optimality:** it returns a **shortest** path
- **Time complexity:** $O(b^d)$ (i.e. exponential in goal depth d)
- **Space complexity:** $O(bd)$ (i.e. linear in goal depth d)

Time and space complexities calculated in **canonical search tree** with branching factor b where **shallowest** goal appears at depth d

Alternative formulation

Each iteration of iterative deepening depth-first search guarantees:

- it visits at least one new node
- it remains optimal (first goal found corresponds to an optimal path)

Instead of increasing the bound one unit at each iteration:

- calculate the depth of the nodes generated but not visited
- set the bound so that next iteration visit those nodes

Iterative deepening depth-first search: alternative

```
1  Node iterative-deepening-depth-first-search():
2      Node root = make-root-node(init())
3      bound = 0
4
5      % perform depth-bounded searches with increasing depth bounds
6      while true
7          pair<Node,unsigned> p = bounded-dfs-visit(root, 0, bound)
8          if p.first != null return p.first
9          bound = p.second
10
11 pair<Node,unsigned> bounded-dfs-visit(Node n, unsigned d, unsigned bound):
12     % base cases
13     if d > bound return (null,d)     % prune branch by depth bound
14     if n.state.is-goal() return (n,d)  % goal is found, return
15
16     % expansion and recursion
17     t = ∞
18     foreach <s,a> in n.state.successors()
19         Node n' = n.make-node(s, a)
20         pair<Node,unsigned> p = bounded-dfs-visit(n', d+1, bound)
21         if p.first != null return p
22         t = min(t,p.second)
23     return (null,t)   % failure: there is no path from node n to goal
```

Iterative deepening uniform-cost search

It is for uniform-cost search what iterative deepening depth-first search is for breadth-first search:

– linear-space algorithm enforcing a **uniform-cost discovery order**

It performs **cost-bounded** depth-first searches, **increasing the cost bound with each iteration** until reaching a goal node

The algorithm is optimal because the increase in cost guarantees:

– it isn't too small: **at least a new node is visited**

– it isn't too large: **no non-optimal goal is found**

The next cost bound is the minimum cost of the nodes **generated but not visited** in the last search

Iterative deepening uniform-cost search: pseudocode

```
1  Node iterative-deepening-uniform-cost-search():
2      Node root = make-root-node(init())
3      bound = 0
4
5      % perform cost-bounded searches with increasing cost bounds
6      while true
7          pair<Node,unsigned> p = cost-bounded-dfs-visit(root, bound)
8          if p.first != null return p.first
9          bound = p.second
10
11 pair<Node,unsigned> cost-bounded-dfs-visit(Node n, unsigned bound):
12     % base cases
13     if n.g > bound return (null,n.g)     % prune branch by depth bound
14     if n.state.is-goal() return (n,n.g)  % goal is found, return
15
16     % expansion and recursion
17     t = ∞
18     foreach <s,a> in n.state.successors()
19         Node n' = n.make-node(s, a)
20         pair<Node,unsigned> p = cost-bounded-dfs-visit(n', bound)
21         if p.first != null return p
22         t = min(t,p.second)
23     return (null,t)   % failure: there is no path from node n to goal
```


Summary

- In general, it is easier to deal with time than to deal with memory
- Depth-first search is incomplete and suboptimal but linear space
- Iterative deepening depth-first search: optimal for unit costs and linear space
- Iterative deepening uniform-cost search: optimal for costs and linear space

Appendix: solving the series $\sum_{k\geq 1} kb^{-k}$

Consider $S_n = \sum_{k=0}^{n} kx^k$ for some constant x

$$\begin{aligned} S_n + (n+1)x^{n+1} &= \sum_{k=0}^{n} kx^k + (n+1)x^{k+1} \\ &= \sum_{k=1}^{n+1} kx^k \;=\; \sum_{k=0}^{n}(k+1)x^{k+1} \\ &= x\sum_{k=0}^{n} kx^k + x\sum_{k=0}^{n} x^k \;=\; xS_n + x\frac{x^{n+1}-1}{x-1} \end{aligned}$$

Solving for S_n, one obtains $S_n = \frac{1}{(1-x)^2}\left[nx^{n+2} + x - (n+1)x^{n+1}\right]$

For $|x| < 1$, take the limit as $n \to \infty$ to get $\sum_{k\geq 1} kx^k = \frac{x}{(1-x)^2}$

Replace $x = b^{-1}$ to get $\sum_{k\geq 1} kb^{-k} = \frac{b}{(1-b)^2}$

Data structures for search algorithms

Goals for the lecture

- Repositories of nodes: open and closed lists
- Data structures to store the open and closed lists
- Time/space guarantees for data structures
- Hash tables to improve efficiency

Algorithmic requirements

Algorithms that store the complete search tree need to store:

– nodes that had been generated and expanded (**closed list**)

– nodes that had been generated but not expanded (**open list**)

– states associated to such nodes

Repositories differ in terms of the **queries** that they must implement

Some algorithms require to maintain additional information for states (e.g. color in breadth-first and uniform-cost search and distances in uniform-cost search)

Closed nodes

Closed node = black node

A node becomes closed when it is expanded

Closed nodes are needed for **path reconstruction**

Algorithms don't perform queries on closed nodes (but, they do on states associated to closed nodes)

There is no need for a data structure to store closed nodes:

Closed nodes stay in memory and are accessible through the parent pointers starting from the open nodes

Open nodes

Open node = gray node

In simplest case, open nodes handled in **first-come first-served** basis

In other cases, open nodes are processed in **order of priority**

We use **queues to order open nodes**: a FIFO queue in the first case, and a priority queue in the second case

In general, the following operations are needed on open nodes:

- **insert/pop** a node from the queue
- **insert/pop** a node from the priority queue
- **decrease the priority** of a node in the priority queue

Open nodes: FIFO queue

It can be implemented with a **linked list** of nodes (i.e. equivalent to augmenting the `Node` data structure with pointers implementing the linked list)

The insert and pop operations require **constant time**, and the space requirement is (small) constant per stored node

Open nodes: priority queue

In simplest case, we need a (min-)priority queue to insert and pop nodes. In other cases, we need to **decrease the priority** of a node

Different data structures for implementing priority queues [CLRS]:

- **binary heap:** stores data in single vector using direct addressing
- **binomial heap:** stores data in trees
- **Fibonacci heap:** stores data in trees

Trade-offs when implementing a priority queue

Procedure	Binary heap (worst-case)	Binomial heap (worst-case)	Fibonacci heap (amortized)
`make-heap`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`insert`	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
`extract-min`	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
`decrease-priority`	$\Theta(\log n)^*$	$\Theta(\log n)$	$\Theta(1)$
`union`	$\Theta(n)$	$O(\log n)$	$\Theta(1)$
`minimum`	$\Theta(1)$	$O(\log n)$	$\Theta(1)$
`delete`	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$

* If an element can be mapped to its position in the heap in $O(\log n)$ time

In binomial and Fibonacci heaps, the asymptotic notation hides "relatively high" constant factors

Other information associated with states

Algorithms need to **query** the color of states (i.e. open/closed status)

Other algorithms need to associate states with an **upper bound** on the cost of a minimum-cost path from the initial state to the state

In general, the following operations are needed on a given state:

– **retrieve/update** the color of the state

– **retrieve/update** the current upper bound for the state

States: naive data structures

Red-black trees [CLRS] can be used to implement **associative arrays** that map states to colors, and states to upper bounds:

```
1  enum { White, Gray, Black } Color   % different colors for states
2
3  map<State, Color> color             % maps states to colors
4  map<State, unsigned> upper_bound    % maps states to upper bounds
```

The time required to access/update these structures is:

– search in $O(\log n)$ time

– insertion in $O(\log n)$ time

– deletion in $O(\log n)$ time

where n is number of states in the data structure

States: hash table

Operations on states can be improved by using a **hash table** instead of red-black trees

Assumption is to have a good **hash function** so that accessing the table takes **constant or sublogarithmic time**

First, `AbstractState` is augmented with fields for color and upper bound, and the hash table is created:

```
1  struct AbstractState {
2      Color color
3      unsigned upper_bound
4
5      bool is_goal()
6      list<pair<Action, State> > successors()
7  };
8
9  typedef hash_table<const AbstractState> HashTable
10 HashTable hash_for_states
```


States: hash table

Second, functions to query/update the hash and fields are provided:

```
1  void insert_in_hash(AbstractState state) {
2      hash_for_states.insert(state)
3  }
4
5  AbstractState find_in_hash(AbstractState state) {
6      return hash_for_states.find(state)
7  }
8
9  Color get_color(AbstractState state) {
10     AbstractState s = find_in_hash(state)
11     return s == null ? White : s.color
12 }
13
14 void set_color(AbstractState state, Color color) {
15     AbstractState s = find_in_hash(state)
16     if s != null {
17         s.color = color
18     } else {
19         state.color = color
20         insert_in_hash(state)
21     }
22 }
```

Can we improve on the priority queue?

Is there a data structure to store the open nodes and with a better performance? Can we have **constant-time** operations?

- In general we don't know how to improve the priority queue
- However, the priorities in search are often integers
- In such cases, **Van Emde Boas** trees [CLRS, 3rd edition] can be used
- Further, the priorities are often bounded by a small constant B which can be exploited to obtain constant-time operations

Summary

- Open and closed lists
- Closed list is not explicitly maintained
- Open list is implemented with a FIFO or priority queue
- Information associated with states is stored in hash tables
- Priority queue can be improved when priorities are small integers