Sztuczna Inteligencja i Systemy Doradcze

Przeszukiwanie przestrzeni stanów

Postawienie problemu

Reprezentacja problemu:

stany: reprezentują opisy różnych stanów świata rzeczywistego
akcje: reprezentują działania zmieniające bieżący stan
koszt akcji (≥ 0): reprezentuje koszt związany z wykonaniem akcji

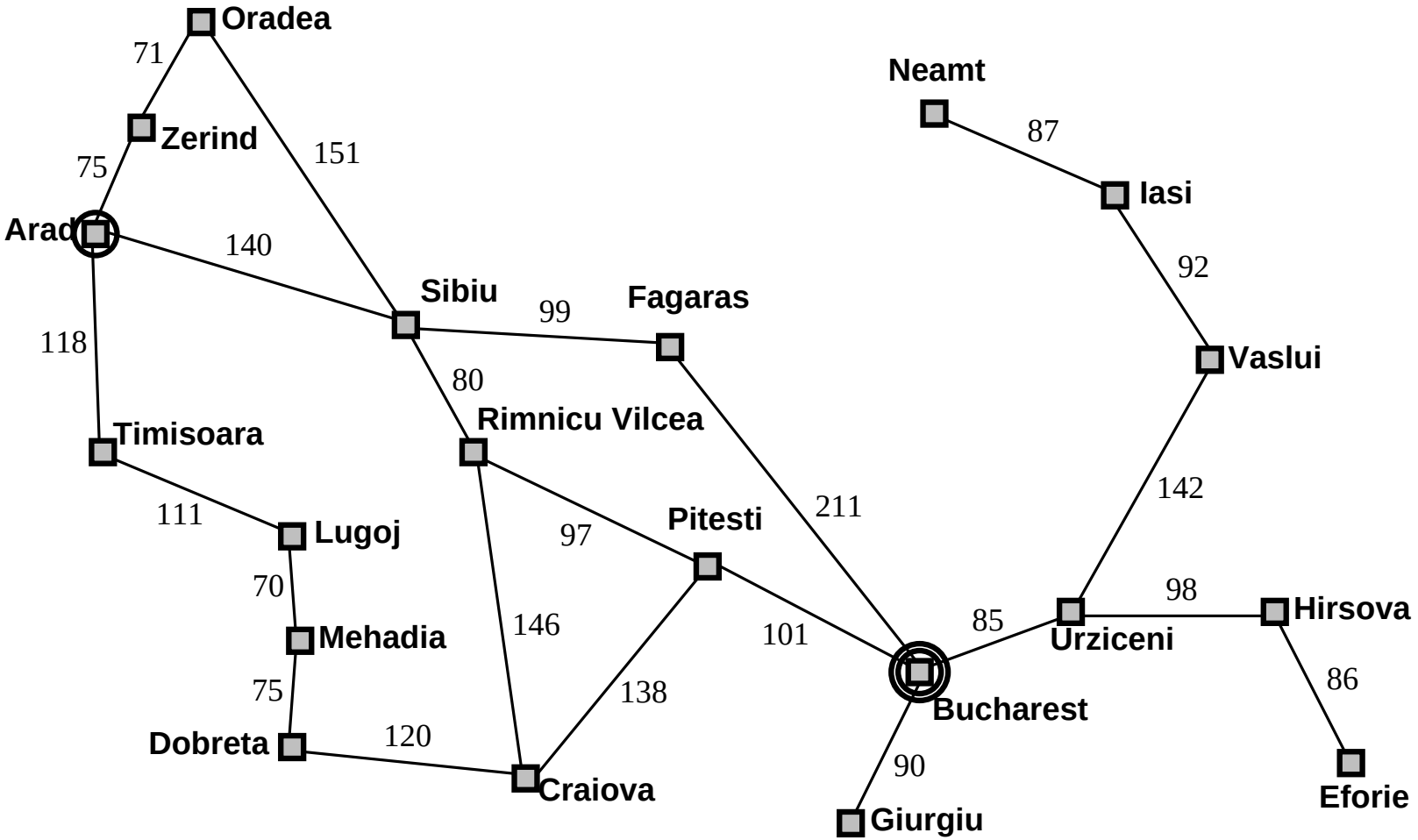
Świat rzeczywisty jest ogromnie złożony: przestrzeń stanów dla problemu musi być wyabstrahowana z rzeczywistości, pojedynczne akcje w opisie problemu muszą reprezentować złożone operacje rzeczywiste

Sformułowanie problemu

stan początkowy: początkowy stan przed rozwiązaniem problemu
cel: stan docelowy lub formuła oceniająca, czy dany stan spełnia cel
rozwiązanie: ciąg akcji prowadzący od stanu początkowego do celu
koszt rozwiązania: funkcja oceny kosztu rozwiązania równa sumie kosztów poszczególnych akcji występujących w rozwiązaniu

Rozwiązania o niższym koszcie są lepsze niż rozwiązania o wyższym koszcie.

Przyklad problemu: Rumunia



Przyklad problemu: Rumunia

Na wakacje do Rumunii; obecnie w Aradzie.
Samolot odlatuje jutro z Bukaresztu

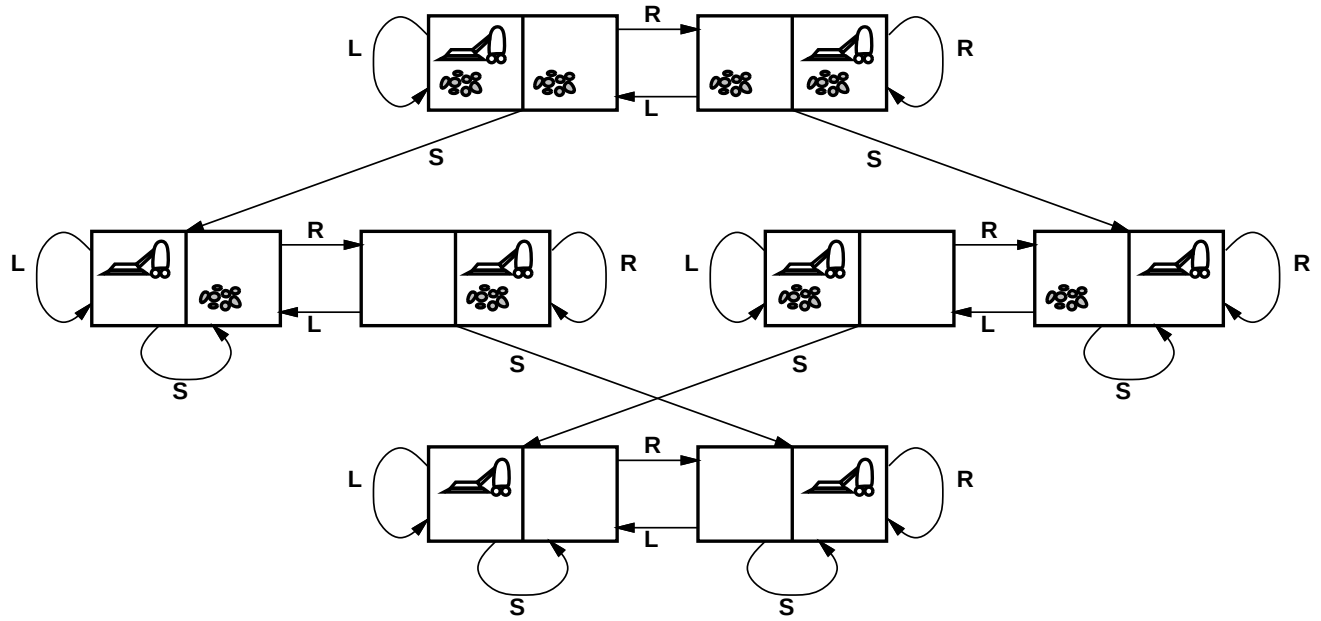
Reprezentacja problemu:

stany: miasta {Arad, Sibiu, Tibisoara, Zerind, ...}
akcje: przejazdy pomiędzy dwoma miastami, np. Arad $\longrightarrow$ Zerind
koszt akcji: odległość pomiędzy dwoma miastami

Sformułowanie problemu

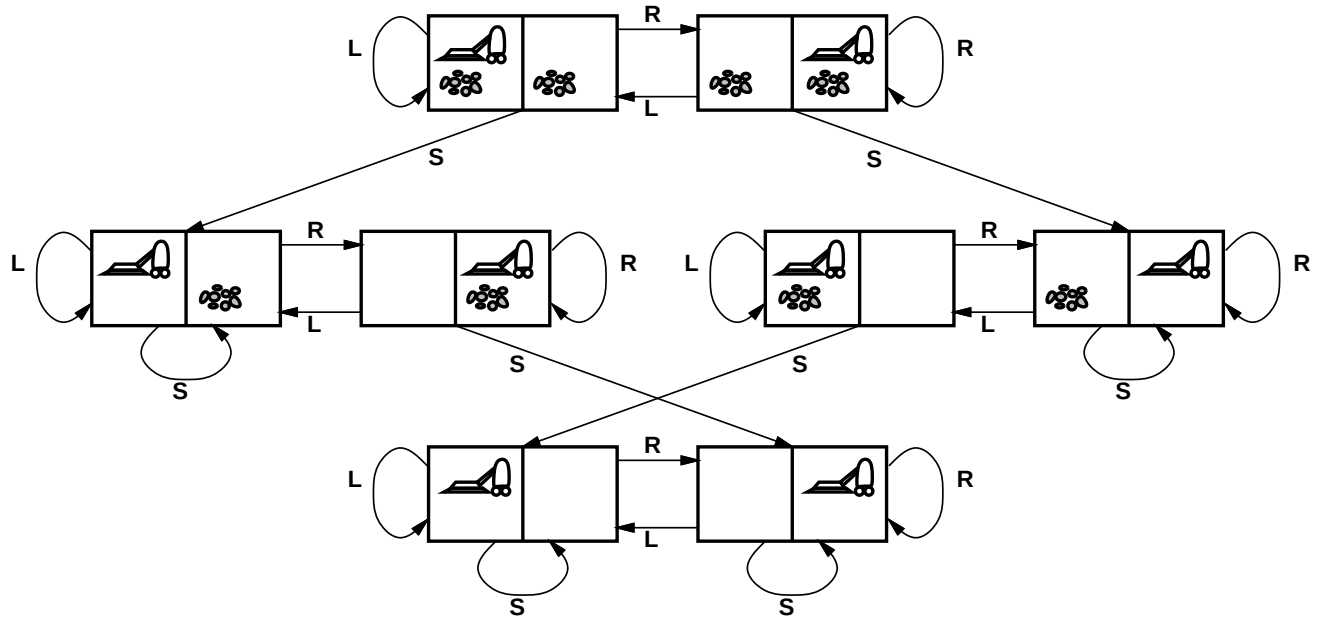
stan początkowy: Arad
stan docelowy: Bukareszt
rozwiązanie: ciąg przejazdów, np. Arad $\longrightarrow$ Sibiu $\longrightarrow$ Fagaras $\longrightarrow$ Bukareszt
koszt rozwiązania: suma kilometrów pomiędzy kolejnymi miastami

Przyklad problemu: sprzatanie



stany??
akcje??
cel??
koszt rozwiązania??

Przyklad problemu: sprzatanie



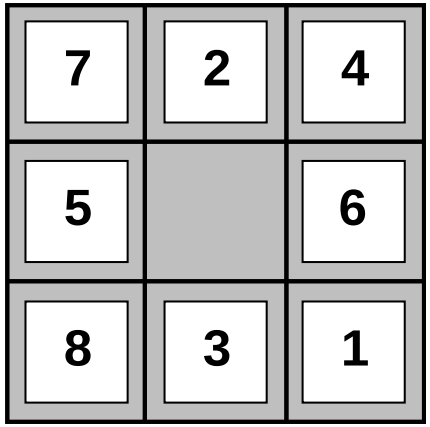
stany??: stan pomieszczeń (czysto/brudno) i lokalizacja robota

akcje??: $Lewo$, $Prawo$, $Odkurzaj$, $NicNieRób$

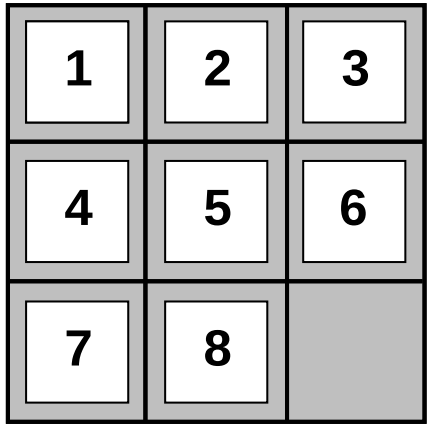
cel??: czysto

koszt rozwiązania??: 1 dla każdej akcji (0 dla $nicNieRób$)

Przyklad problemu: 8-elementowe puzzle



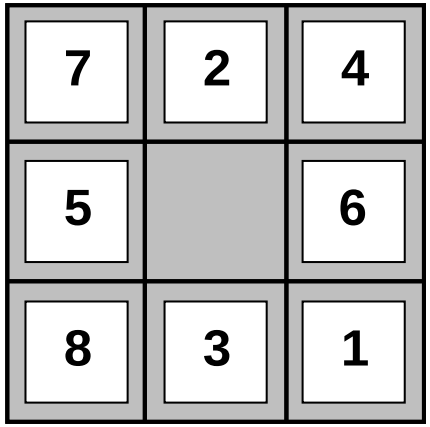
Start State



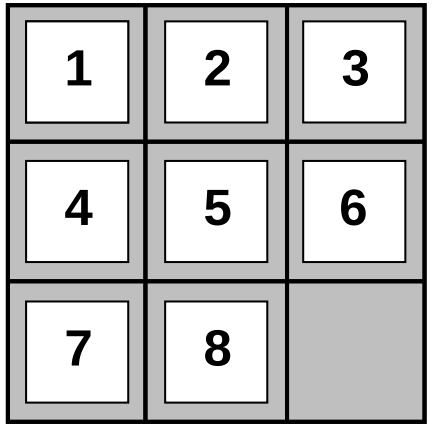
Goal State

stany??
akcje??
cel??
koszt rozwiązania??

Przyklad problemu: 8-elementowe puzzle



Start State

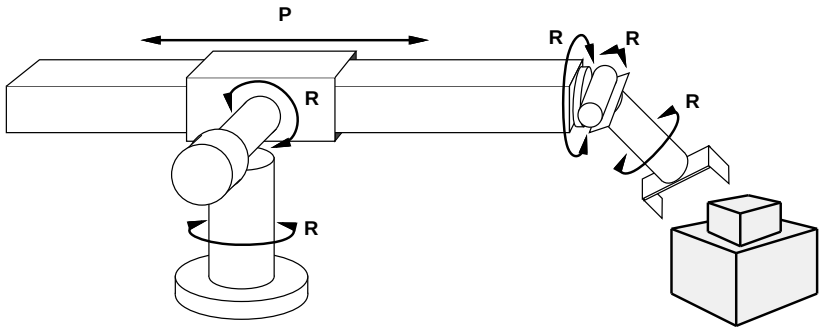


Goal State

stany??: rozmieszczenia puzzli
akcje??: przesuń puste miejsce w prawo, w lewo, w górę, w dół
cel??: wybrane rozmieszczenie puzzli
koszt rozwiązania??: 1 za każdy ruch

[Uwaga: znalezienie optymalnego rozwiązania dla rodziny problemów n-elementowych puzzli jest NP-trudne]

Przyklad problemu: montaz przy uzyciu robota



stany??: rzeczywiste współrzędne kątów w złączeniach robota
elementy do zmontowania

akcje??: ciągłe ruchy złączy robota

cel??: kompletny montaż (*ale nie robota!*)

koszt rozwiązania??: czas montażu

Przeszukiwanie drzewa stanow

Prosty pomysł:

symulowanie offline przeszukiwania przestrzeni stanów

poprzez generowanie następników wcześniej odwiedzonych stanów

(znane również jako *ekspansja* stanów)

function TREE-SEARCH(*problem, strategy*) **returns** a solution, or failure

initialize the search tree using the initial state of *problem*

loop do

if there are no candidates for expansion **then return** failure

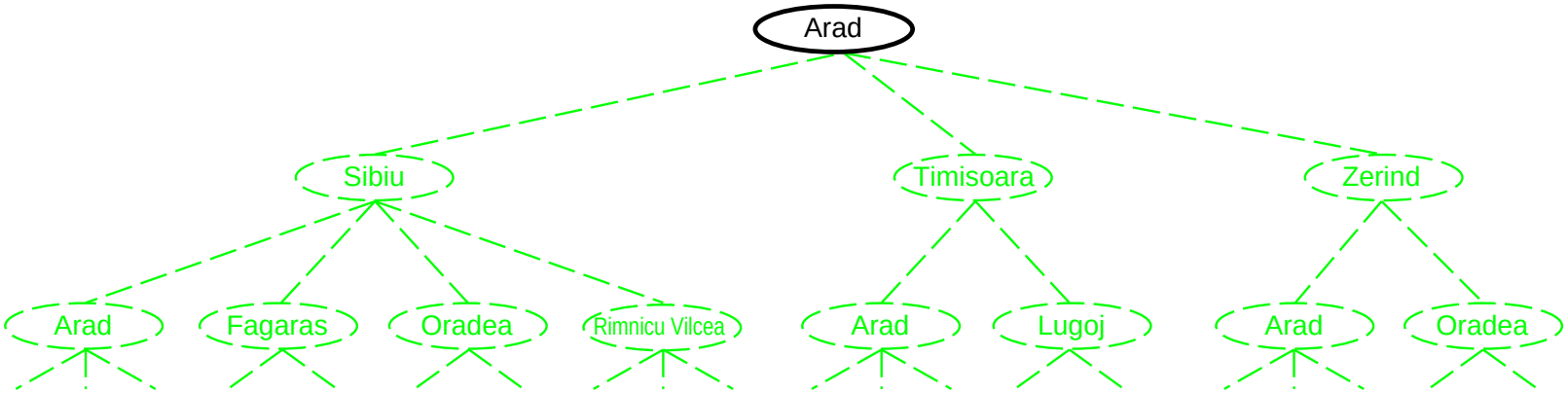
choose a leaf node for expansion according to *strategy*

if the node contains a goal state **then return** the corresponding solution

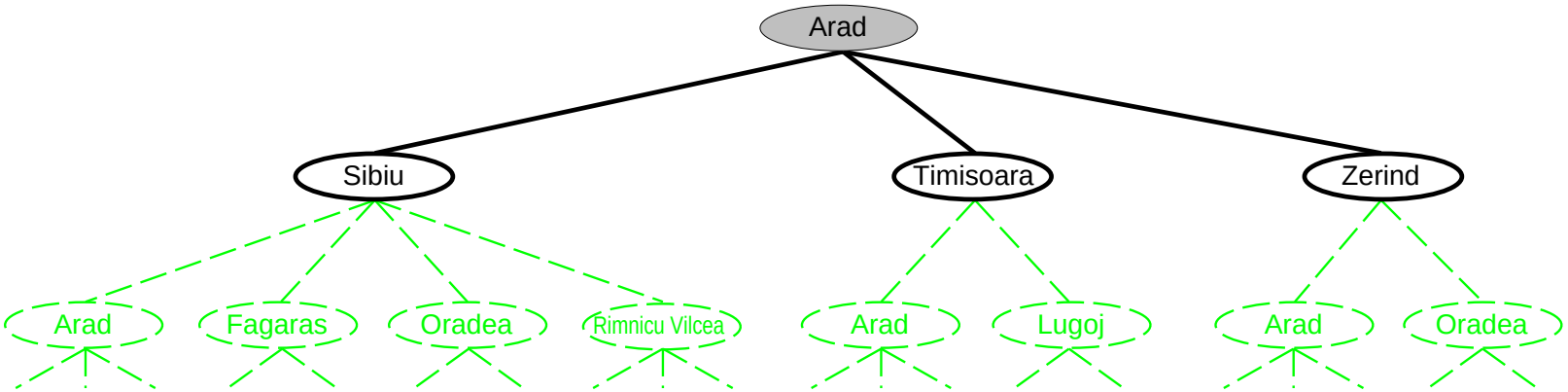
else expand the node and add the resulting nodes to the search tree

end

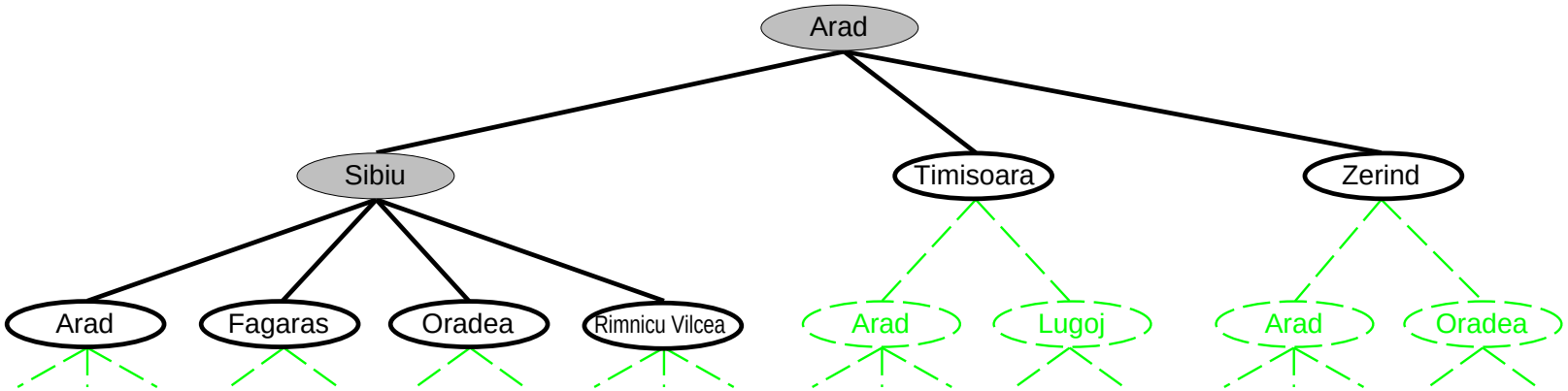
Przyklad przeszukiwania drzewa stanow



Przyklad przeszukiwania drzewa stanow



Przyklad przeszukiwania drzewa stanow



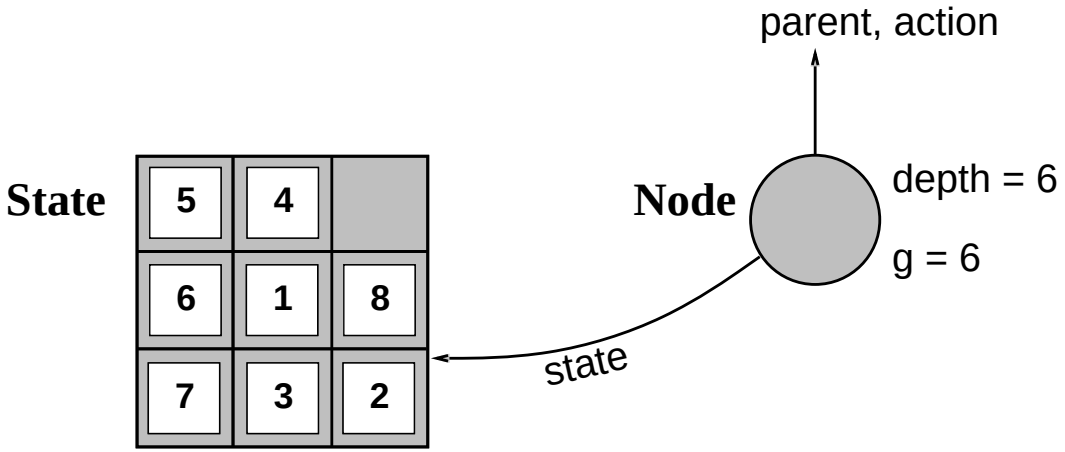
Implementacja: stany vs. wezly

Stan jest fizyczną konfiguracją (jej reprezentacją)

Węzeł jest strukturą danych stanowiącą część drzewa przeszukiwań

zawiera *poprzednik* $(parent)$, *następniki*, *głębokość* $(depth)$ oraz *koszt ścieżki od korzenia* (g)

Stany nie mają poprzedników, następników, głębokości i kosztu ścieżki!



Funkcja SUCCESSORFN zwraca jako wynik zbiór akcji możliwych do wykonania w danym stanie wraz ze stanami osiąganymi po wykonaniu akcji.

Funkcja EXPAND tworzy nowe węzły i wypełnia ich pola używając funkcji SUCCESSORFN.

Implementacja: przeszukiwanie drzewa stanow

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem] applied to STATE(node) succeeds return node
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

Strategie przeszukiwania

Strategia jest definiowana poprzez wybór *kolejności ekspansji stanów*

Strategie są ocenianie według następujących kryteriów:

- zupełność — czy zawsze znajduje rozwiązanie, jeśli ono istnieje?
- złożoność czasowa — liczba wygenerowanych węzłów
- złożoność pamięciowa — maksymalna liczba węzłów w pamięci
- optymalność — czy znajduje rozwiązanie o minimalnym koszcie?

Złożoność czasowa i pamięciowa są mierzone w terminach

- b — maksymalnego rozgałęzienia drzewa przeszukiwań
- d — głębokości rozwiązania o najmniejszym koszcie
- m — maksymalnej głębokości drzewa przeszukiwań (może być ∞)

Rodzaje strategii przeszukiwania

Strategie *ślepe* korzystają z informacji dostępnej jedynie w definicji problemu:

- ♢ Przeszukiwanie wszerz
- ♢ Strategia jednolitego kosztu
- ♢ Przeszukiwanie wgłąb
- ♢ Przeszukiwanie ograniczone wgłąb
- ♢ Przeszukiwanie iteracyjnie pogłębiane
- ♢ Przeszukiwanie dwukierunkowe

Strategie *heurystyczne* korzystają z dodatkowej, heurystycznej funkcji oceny stanu (np. szacującej koszt rozwiązania od bieżącego stanu do celu):

- ♢ Przeszukiwanie zachłanne
- ♢ Przeszukiwanie A*
- ♢ Rekurencyjne przeszukiwanie pierwszy najlepszy
- ♢ Przeszukiwanie lokalne zachłanne (hill-climbing)
- ♢ Symulowane wyżarzanie
- ♢ Algorytm genetyczny