Programming Assignment 1 – Problem Solving Using Search

Sean G Meyer

Problem Description

The goal of this program is to implement a series of programs to solve sliding-tile puzzles using various search methods. For the purposes of this problem the sliding tile puzzle is limited to n x n puzzles, where n $>$ 1. These puzzles are known as $< n^2 - 1 >$ puzzles, with some of the most common being 8-puzzles and 15-puzzles.

The puzzles will be read in through a JSON input which includes the n value, the start state, and the end state. This JSON should be validated to ensure it is correct. The specifics of this are under Part 1 below.

Possible rules for the puzzle game will be generated (such as up, left, down, right), and the applicable rules for a given state will be used to generate successor states. In this way, a starting state leads to all possible successor states, and these rules can be used to apply this transition across different search methods (see Part 2).

The specific searches to be tested are: Depth-First Search with Backtracking Control, Iterative Deepening using Backtracking Control, Graph Search, and Algorithm A*. The specifics of these are listed under Part 3 and Part 4 below.

Part 1

The first thing that needed to be accomplished was reading in and validating game JSON files. My code does this by including a JSON library, reading in the file, and then converting the JSON to a native Clojure data structure.

Then using this data structure, checks can be done against the structure to ensure that it is valid. The pieces of my code that handle this aspect are show below.

```
(:require [clojure.data.json :as json])

(def game-json)

(defn validate-json
  [game]
  (cond
    (not (contains? game :n))
    (do (println "json does not contain 'n' key") false)
    (not (contains? game :start))
    (do (println "json does not contain 'start' key") false)
    (not (contains? game :goal))
    (do (println "json does not contain 'goal' key") false)
    (not (> (:n game) 1))
    (do (println "value of 'n' in json must be greater than 1")
    ↪  false)
    :else true))

(defn -main
  [json-file]
  (def game-json
    (json/read-str (slurp json-file)
                   :key-fn keyword))
  (when (validate-json game-json)
    ;;run searches)
```

Part 2

The second part of this problem was determining which rules were applicable for a given state, and returning a list of those rules along with the capability to use the rules to return a the successor state for a given state/rule.

I chose to simplify things for myself a bit here and combine the rules and transition model into one function, so that the rules and associated successor states are both returned together. It wouldn't be a big challenge to separate these out, but in the case of this problem I felt it made the code a bit clearer. The functions for achieving this are shown below (a function to get position of an element, a function to perform a move, and a function which finds valid actions and performs them).

```
(defn get-position
  "Returns a map with keys :x and :y representing position of
  ↪  element within matrix"
  [elem matrix]
  (first
    (for [[y row] (map-indexed vector matrix)
          [x val] (map-indexed vector row)
          :when (= elem val)]
         {:x x :y y})))
```

```clojure
(defn perform-move
  "Return board with the 0 tile moved"
  [state cur-pos new-pos]
  (swap! boards-generated inc)
  (let [moved-0 (assoc state (:y new-pos)
                            (assoc (state (:y new-pos))
                              (:x new-pos) 0))]
    (assoc moved-0 (:y cur-pos)
                  (assoc (moved-0 (:y cur-pos))
                    (:x cur-pos)
                    ((state (:y new-pos)) (:x new-pos))))))

(defn do-actions
  "Return list of applicable rules and their resulting successor
  ↪  states"
  [state]
  (let [pos (get-position 0 state)
        x (:x pos)
        y (:y pos)]
    (remove #(nil? %)
      (conj ()
        (when (< (inc y) (:n game-json))
          {:action :down, :new-state (perform-move state pos {:x x
           ↪  :y (inc y)})})
        (when (> y 0)
          {:action :up, :new-state (perform-move state pos {:x x :y
           ↪  (dec y)})})
        (when (< (inc x) (:n game-json))
          {:action :right, :new-state (perform-move state pos {:x
           ↪  (inc x) :y y})})
        (when (> x 0)
          {:action :left, :new-state (perform-move state pos {:x
           ↪  (dec x) :y y})})))))
```

Part 3

This involved implementing depth-first search with a backtracking control strategy, and a depth bound. The Psuedo-Code for this was supplied, and my code is mostly a direct translation (with some tweaks due mostly to the immutable nature of Clojure data structures). This code is used for both the standard depth-first search and for iterative deepening.

```
(defn backtrack1
  [datalist bound]
  (let [data (first datalist)]
    (cond
      (.contains (rest datalist) data) :fail
      (match-goal? (:new-state data)) nil
      (> (count datalist) bound) :fail
      :else
      (loop [successors (do-actions (:new-state data))]
        (if (empty? successors)
          :fail
          (let [rdata (first successors)
                successors (rest successors)
                rdatalist (cons rdata datalist)
                path (backtrack1 rdatalist bound)]
            (if (= path :fail)
              (recur successors)
              (cons rdata path)))))))))
```

Part 4

Part 4 was the implementation of graph search. I specifically implemented Uniform-Cost-Search from the book (p. 84) as it was a bit easier and valid in this particular case (since the triangle inequality holds for the sliding puzzles).

Working with a priority-queue structure for the "frontier" (as it is described in the book) turned out to be relatively difficult for me to implement in Clojure, so my specific implementation does deviate somewhat in structure from the psuedo code in the text. The algorithm itself is the same though, but I do believe my inexperience with lisps and functional coding led to much more complex code than strictly necessary.

```
(defn uniform-cost-search
  [initial-state goal-test? heuristic-fn]
  (loop [node {:state initial-state :parent nil :path-cost 0
  ↪  :total-cost (heuristic-fn initial-state) :action nil}
         explored {}
         frontier (priority-map-keyfn :total-cost (:state node)
          ↪  node)]
    (cond
      (empty? frontier) :failure
      (goal-test? (:state node)) node
      :else (let [frontier
                  (reduce (fn [frontier result]
                            (cond
                              (and (not (contains? frontier
                              ↪   (:new-state result)))
                                   (not (contains? explored
                                   ↪   (:new-state result))))
                              (conj frontier {(:new-state result)
                              ↪   (create-new-frontier-node result
                              ↪   node heuristic-fn)})
                              (and (contains? frontier (:new-state
                              ↪   result))
                                   (> (+ (inc (:path-cost node))
                                         (heuristic-fn (:new-state
                                         ↪   result)))
                                      (:total-cost (get frontier
                                      ↪   (:new-state result)))))
                              (assoc frontier (:new-state result)
                              ↪   (create-new-frontier-node result
                              ↪   node heuristic-fn))
                              :else frontier))
                          frontier (do-actions (:state node)))]
              (recur
                (val (peek frontier))
                (conj explored {(:state node) node})
                (pop frontier))))))

(defn create-new-frontier-node
  [result node heuristic-fn]
  {:state (:new-state result)
          :parent node
          :path-cost (inc (:path-cost node))
          :total-cost (+ (inc (:path-cost node))
                         (heuristic-fn (:new-state result)))
          :action (:action result)})

(defn uniform-cost-heuristic
  [state]
  0)

(defn match-goal?
  [cur-state]
  (= cur-state (:goal game-json)))
```

Part 5

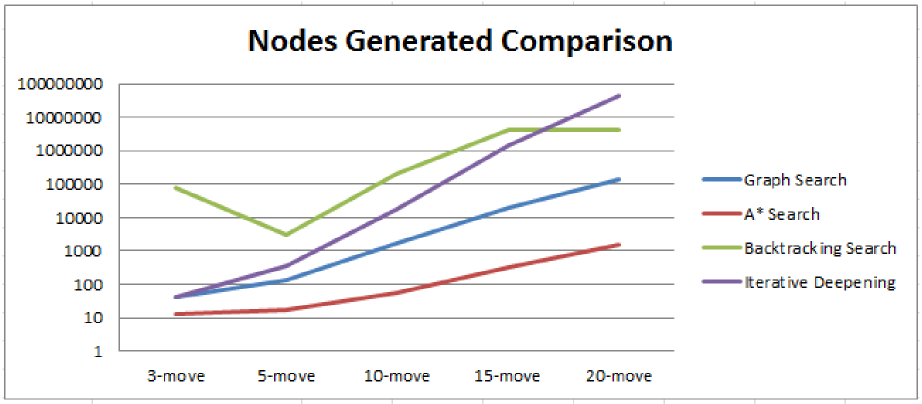
The next task was to implement algorithm A*, specifically using the Manhattan Distance and number of misplaced tiles as heuristic functions. Beyond the code to calculate the heuristic based on a given state, there were no modifications to the graph search from Part 4. All that needed to be done was to pass in the new heuristic function to the graph search call.

```
(defn distance-between-points
  [x1 x2 y1 y2]
  (+ (Math/abs (- x1 x2)) (Math/abs (- y1 y2))))

(defn get-distance-for-element
  [elem start-state end-state]
  (let [pos1 (get-position elem start-state)
        pos2 (get-position elem end-state)]
    (distance-between-points (:x pos1) (:x pos2) (:y pos1) (:y
    ↪  pos2))))

(defn manhattan-distance
  [state]
  (let [end-state (:goal game-json)
        max (dec (* (:n game-json) (:n game-json)))]
    (reduce #(+ %1 (get-distance-for-element %2 state end-state))
            0 (range 1 max))))

(defn a*-heuristic
  [state]
  (manhattan-distance state))
```

Results

All four searches were ran on various puzzles with different size and solution depth, keeping count of the board states generated as well as the solution path.

Below is a graph of the number of nodes generated for each search type on the various puzzles, using a logarithmic scale (backtracking was run with a depth bound of 26 for all searches).



Below is a condensed and shortened list of results (printing everything gives a huge amount of text).

```
10-moves.json
-------------
Graph Search
Boards Generated:  1658
Solution Length:  10

A* Search
Boards Generated:  53
Solution Length:  10

Backtracking Search
Boards Generated:  196865
Solution Length:  26

Iterative Deepening Search Using Backtracking
Boards Generated:  18219
Solution Length:  10


15-moves.json
-------------
Graph Search
Boards Generated:  20023
Solution Length:  15

A* Search
Boards Generated:  326
Solution Length:  15
```

```
Backtracking Search
Boards Generated:  4149710
Solution Length:  25

Iterative Deepening Search Using Backtracking
Boards Generated:  1413146
Solution Length:  15


20-moves.json
-------------
Graph Search
Boards Generated:  133010
Solution Length:  20

A* Search
Boards Generated:  1540
Solution Length:  20

Backtracking Search
Boards Generated:  4143326
Solution Length:  26

Iterative Deepening Search Using Backtracking
Boards Generated:  44095242
Solution Length:  20
```

The two views above show the short results where just the length of the solution is given, but these algorithms also enable a list of actions taken to reach the goal state. For brevity's sake I'll include just one such result below. This is specifically the result of running A* on the 26-move puzzle.

```
A* Search - 26-Move Puzzle
----------
Boards Generated -  15117
Solution Length -  26
{:action :left, :new-state [[7 2 4] [0 5 6] [8 3 1]]}
{:action :up, :new-state [[0 2 4] [7 5 6] [8 3 1]]}
{:action :right, :new-state [[2 0 4] [7 5 6] [8 3 1]]}
{:action :down, :new-state [[2 5 4] [7 0 6] [8 3 1]]}
{:action :right, :new-state [[2 5 4] [7 6 0] [8 3 1]]}
{:action :down, :new-state [[2 5 4] [7 6 1] [8 3 0]]}
{:action :left, :new-state [[2 5 4] [7 6 1] [8 0 3]]}
```

```
{:action :left, :new-state [[2 5 4] [7 6 1] [0 8 3]]}
{:action :up, :new-state [[2 5 4] [0 6 1] [7 8 3]]}
{:action :right, :new-state [[2 5 4] [6 0 1] [7 8 3]]}
{:action :right, :new-state [[2 5 4] [6 1 0] [7 8 3]]}
{:action :down, :new-state [[2 5 4] [6 1 3] [7 8 0]]}
{:action :left, :new-state [[2 5 4] [6 1 3] [7 0 8]]}
{:action :left, :new-state [[2 5 4] [6 1 3] [0 7 8]]}
{:action :up, :new-state [[2 5 4] [0 1 3] [6 7 8]]}
{:action :right, :new-state [[2 5 4] [1 0 3] [6 7 8]]}
{:action :right, :new-state [[2 5 4] [1 3 0] [6 7 8]]}
{:action :up, :new-state [[2 5 0] [1 3 4] [6 7 8]]}
{:action :left, :new-state [[2 0 5] [1 3 4] [6 7 8]]}
{:action :left, :new-state [[0 2 5] [1 3 4] [6 7 8]]}
{:action :down, :new-state [[1 2 5] [0 3 4] [6 7 8]]}
{:action :right, :new-state [[1 2 5] [3 0 4] [6 7 8]]}
{:action :right, :new-state [[1 2 5] [3 4 0] [6 7 8]]}
{:action :up, :new-state [[1 2 0] [3 4 5] [6 7 8]]}
{:action :left, :new-state [[1 0 2] [3 4 5] [6 7 8]]}
{:action :left, :new-state [[0 1 2] [3 4 5] [6 7 8]]}
```

Conclusions

It's immediately apparent from the results that informed search algorithms vastly outperform uninformed ones (given a decent heuristic function). The logarithmic graph really shows off just how small the number of boards generated is when using A* compared to all of the uninformed searches.

Beyond that, graph search (in this case breadth-first-search) performed surprisingly well. I would conclude that it is a relatively good choice when the problem is small enough to fit the entire state space in memory and there is no better information available to use something like A*.

Both of the depth first searches performed quite poorly in comparison, with iterative deepening growing extremely large as the solution depth increased. While this search may be necessary when memory is a limiting factor, I would not reach for it unless specifically necessary.