競技問題 (NUPSC 2018)

遠い遠いところに「ネコ星」という惑星があります。 その惑星の地上を治めているのは猫という生物でした。 その生物は高度な知能を持っていますが、 常に怠惰を美徳としています。

ネコ星には様々な国もあります。 その中に「ニャポン国」という国がありました。 ニャポン国はネコ星の中一番新参の国でした。 但しその住民たちはそれはそれは聡明なネコ星の住民よりも又一段と賢く、 特に土木技術に長けていました。

ニャポン国の国民達は土木技術に長けているのは実はわけがあります。 それはニャポン国は一年ごと全ての都市を取り壊し、 全く違う場所で新しい都市を作る慣習があるからです。 我々には理解しがたい慣習ではありますが、 ニャポン国建国の父らが言わく「猫たるものは己の知性を生かし、これを誇るべし」と。 ニャポン国の大統領は都市の再建は建国の父らのその教えの表しだと宣言しています。 どうやら財産と労力をひたすら無駄に使う猫たちの国のようです。 それはそれも仕方ありません。 何故なら財産と労力を意味のあるところに使うのであれば、 それは怠惰ではありません。 ネコ星の猫たちはあくまでも怠惰を美徳としていますから。

さて、何の事由があったかは知りませんが、 君は何故かそのニャポン国の技術顧問に雇われてしまいました。 君に振られた仕事はニャポン国その壮大な都市再建計画のスマート化です。 勿論スマート化とは情報技術を使い、従来の仕事を効率良くすることです。 我々の地球と同じく、「スマート」という単語はネコ星でもバズワードだそうです。

プロジェクト1

ニャポン国の都市の場所は毎年変っていますので毎年新しい道路交通網も再建しないといけないわけになります。 ニャポン国の猫たちは聡明ではありますが、 あくまでも怠惰を美徳としています。 ですので一メートルたりとも無駄に道路を作りたくない。 君はそんなニャポン国のために始めてしてあげる仕事は、 道路交通網を設計するプログラムを組むことにしました。 又道路交通網を設計するついでに、 ナビシステムも組むことにしました。

タスク1a

以下のプログラムを修正し、 地図にある全ての都市の名前を枚挙する関数 `list-cities` を完成させなさい。 但しその関数の引数 `m` は以下の形式のビットマップを取り、 出力(つまり関数の返り値)は都市名のシーケンスでなければならない。

この関数を正しく完成させた者は100点を得ます。

`m` は平面上の地図を表わします。 `(m :width)` はその地図の幅、 `(m :height)` はその地図の長さを表わしています。 それぞれの点 (x, y) では `(m [x y])` はその点にある都市名を表します。 但し `:void` は都市名ではなく、 空地を意味しています。 例えばサンプル入力では `(m [2 1])` は `:nyagoya` となっていまして、 `(m [2 2])` は `:void` になっています。 それぞれ (2, 1) に「nyagoya」という都市があることと、 (2, 2) に都市が存在しないことを表わしています。

```
(defn list-cities [m]
  (for [x (range (m :width))
        y (range (m :height))
        :when (not (= (m [x y]) :void))]
    (m [x y])))
```

```
#'cljs.user/list-cities
```

タスク1b

以下のプログラムを修正し、 道路交通網の設計関数 `plan-road-system` を完成させなさい。 但しその関数の引数 `m` はタスク1aと同じような地図を表わすビットマップを取り、 出力(つまり関数の返り値)は都市名を頂点、道路を辺として持つグラフでなければならない。 又出力のグラフの辺のウェイトはその道路の長さでなければならない。 長さは都市の位置から割り出すユークリッド距離でなければならない。 例えば (1, 7) にある都市と (9, 4) にある都市とを繋ぐ道路の長さは $\sqrt{(1-7)^2+(9-4)^2}$ である。

この関数を正しく完成させた者は50点を得ます。 又完成させた `plan-road-system` が設計した道路交通網の全長を用いてプログラムの優劣を評価し、 最も優れたプログラムを完成させた者は追加で50点を得ます。 その他この関数を正しく完成させた者は優劣順位に従い、追加で1-49点を得ます。

```clojure
(defn dist [a b]
  (let [sq (fn [x] (* x x))
        [x1 y1] a
        [x2 y2] b]
    (Math/sqrt (+ (sq (- x1 x2))
                  (sq (- y1 y2))))))
(defn enum-all-edges [cities]
  (for [[c1 p1] cities
        [c2 p2] cities
        :when (not= c1 c2)]
    [c1 c2 (dist p1 p2)]))
(defn cities-comp-graph [cities]
  (reduce (fn [g [a b weight]]
            (add-edge g a b weight))
          (empty-graph)
          (enum-all-edges cities)))
(defn list-cities [m]
  (for [x (range (m :width))
        y (range (m :height))
        :when (not (= (m [x y]) :void))]
    [(m [x y]) [x y]]))
(defn plan-road-system [m]
  (minimal-spanning-tree
    (cities-comp-graph (list-cities m))
    identity))
```

```
#'cljs.user/plan-road-system
```

タスク1c

以下のプログラムを修正し、 都市から都市まで移動できる経路を計算する関数 `city-navigation` を完成させなさい。 但しその関数の引数 `g` はタスク1bの出力と同じようなグラフ、 引数 `from` と引数 `to` はそれぞれ出発地の都市名と目的地の都市名を取り、 出力(つまり関数の返り値)は出発地と目的地を含む経過する都市の都市名のシーケンスでなければならない。

この関数を正しく完成させた者は50点を得ます。 又完成させた `city-navigation` が割り出した経路の長さを用いてプログラムの優劣を評価し、 最も優れたプログラムを完成させた者は追加で50点を得ます。 その他この関数を正しく完成させた者は優劣順位に従い、追加で1-49点を得ます。

```clojure
(defn city-navigation [g from to]
  (shortest-path g identity from to))
```

```
#'cljs.user/city-navigation
```

プロジェクト2

ニャポン国の都市や道路交通網は毎年変っていますので、 その住民は自分の居場所を正確に把握して友達や家族に伝えるのはとても難しいようです。 この状況を改善するために君は自動的に住民たちの居場所をトラッキングするシステムを作ることにしました。

注：タスク2aに進む前タスク2sの得点説明をまず読むことをおすすめします。

タスク2a

トラッキングするための状態空間を猫の名前から猫の位置に写すマップとする。 例えば `{:taro [2 4], :haru [3 2]}` をもって、 `:taro` が (2, 4) の位置にいて、 `:haro` は (3, 2) の位置にいる状態を表すことにする。

以下のプログラムを修正し、 `[:move` *猫の名前* *移動先*`]` の形のコマンドを引数 `cmd` で受け取り、 対応する状態遷移関数を返す関数 `compile-move` を完成させなさい。

この関数を正しく完成させた者は20点を得ます。

```
(defn compile-move [[cmd cat moveto]]
  (fn [prev]
    (assoc prev cat moveto)))
```

```
#'cljs.user/compile-move
```

タスク2b

以下のプログラムを修正し、 `{:nyagoya [2 4], :myae [4 7]}` のように 都市名から位置に写すマップ `cities` と、 `[:move-city` *猫の名前* *移動先の都市名*`]` の形のコマンドを引数 `cmd` で受け取り、、 対応する状態遷移関数を返す関数 `compile-move-city` を完成させなさい。

```
(defn compile-move-city [cities [cmd cat moveto]]
  (let [moveto-pos (cities moveto)]
    (fn [prev]
      (assoc prev cat moveto-pos))))
```

```
#'cljs.user/compile-move-city
```

この関数を正しく完成させた者は40点を得ます。

タスク2c

猫たちの健康管理のため、 君は猫たちの移動距離も同じシステムでトラッキングすることにしました。

従ってトラッキングするための状態空間を少し拡張して、 猫の名前から、猫の位置

と猫が移動した距離を記録したマップに写すマップとする。 例えば `{:taro {:pos [2 4], :moved 10}, :haru {:pos [3 2], :moved 20}}` をもって、 `:taro` が $(2, 4)$ の位置にいて、合計10メートルを移動したことがあり、 `:haro` は $(3, 2)$ の位置にいて、合計20メートルを移動したことがあるという状態を表すことにする。

以下のプログラムを修正し、 `{:nyagoya [2 4], :myae [4 7]}` のように 都市名から位置に写すマップ `cities` と、 `[:move` *猫の名前* *移動先*`]` か `[:move-city` *猫の名前* *移動先の都市名*`]` の形のコマンドを引数 `cmd` で受け取り、 対応する状態遷移関数を返す関数 `compile-moving` を完成させなさい。

但し移動距離の計算は都市のある位置からと都市のある位置に移動した場合ユークリッド距離で計上し、 そうでない場合(出発地か目的地のいずれが荒野)マンハッタン距離で計上しなければならない。

この関数を正しく完成させた者は60点を得ます。

```
(defn dist-euc [[x1 y1] [x2 y2]]
  (let [sq (fn [x] (* x x))]
    (Math/sqrt (+ (sq (- x1 x2))
                  (sq (- y1 y2))))))
(defn dist-man [[x1 y1] [x2 y2]]
  (let [abs #(if (≥ % 0) % (- %))]
    (+ (abs (- x1 x2))
       (abs (- y1 y2)))))
(defn reverse-map [m]
  (reduce (fn [acc [k v]] (assoc acc v k)) {} m))
(defn move-dist [rcities from to]
  (if (and (rcities from) (rcities to))
    (dist-euc from to)
    (dist-man from to)))
(defn cat-updater [rcities cat pos]
  (fn [prev]
    (let [{pos0   :pos
           moved0 :moved} (prev cat)]
      (assoc prev cat
        {:pos pos
         :moved (+ moved0 (move-dist rcities pos0
pos))}))))
(defn compile-moving [cities [cmd cat moveto]]
  (let [rcities (reverse-map cities)]
    (if (= cmd :move)
      (cat-updater rcities cat moveto)
      (cat-updater rcities cat (cities moveto)))))
```

```
#'cljs.user/compile-moving
```

タスク2s

以下のプログラムを修正し、猫たちの移動をトラッキングする関数 `track-moving` を完成させなさい。但し引数 `cities` を `{:nyagoya [2 4], :myae [4 7]}` のように都市名から位置に写すマップ、引数 `cmds` を `[:move` *猫の名前* *移動先*`]` か `[:move-city` *猫の名前* *移動先の都市名*`]` の形のコマンドのシーケンス、引数 `cat` を結果を問う猫の名前とする。関数 `track-moving` は `cmds` のような動きが記録された後、猫 `cat` の最終位置と合計移動距離を `{:pos` *最終位置*`, :moved` *合計移動距離*`}` のようなマップで返さなければならない。又全ての猫の最初位置は $(0,0)$ とすし、移動距離の計算は都市のある位置からと都市のある位置に移動した場合ユークリッド距離で計上し、そうでない場合(出発地か目的地のいずれが荒野)マンハッタン距離で計上しなければならない。

```
(defn dist-euc [[x1 y1] [x2 y2]]
  (let [sq (fn [x] (* x x))]
    (Math/sqrt (+ (sq (- x1 x2))
                  (sq (- y1 y2))))))
(defn dist-man [[x1 y1] [x2 y2]]
  (let [abs #(if (≥ % 0) % (- %))]
    (+ (abs (- x1 x2))
       (abs (- y1 y2)))))
(defn reverse-map [m]
  (reduce (fn [acc [k v]] (assoc acc v k)) {} m))
(defn move-dist [rcities from to]
  (if (and (rcities from) (rcities to))
    (dist-euc from to)
    (dist-man from to)))
(defn cat-updater [rcities cat pos]
  (fn [prev]
    (let [{pos0   :pos
           moved0 :moved} (prev cat)]
      (assoc prev cat
        {:pos pos
         :moved (+ moved0 (move-dist rcities pos0
pos))}))))
(defn compile-moving [cities [cmd cat moveto]]
  (let [rcities (reverse-map cities)]
    (if (= cmd :move)
      (cat-updater rcities cat moveto)
      (cat-updater rcities cat (cities moveto)))))

(defn track-moving [cities cmds cat]
  (let [compiled (map #(compile-moving cities %) cmds)
        cats (for [[cmd cat moveto] cmds] cat)]
    (loop [prev (reduce #(assoc % %2 {:pos [0 0]
                                      :moved 0})
                        {cat {:pos [0 0]
                              :moved 0}}
                        cats)
           updaters compiled]
      (if (empty? updaters)
        (prev cat)
        (recur ((first updaters) prev)
               (rest updaters))))))
```

```
#'cljs.user/track-moving
```

この関数を正しく完成させた者は200点を得ます。

但し、タスク2sにおける得点がタスク2a-2cにおける得点より多い場合、 タスク2a-2cの得点は **無効** となり、 タスク2sの得点のみを評価します。 逆にタスク2sにおける得点がタスク2a-2cにおける得点より少い場合、 タスク2sの得点は無効となります。