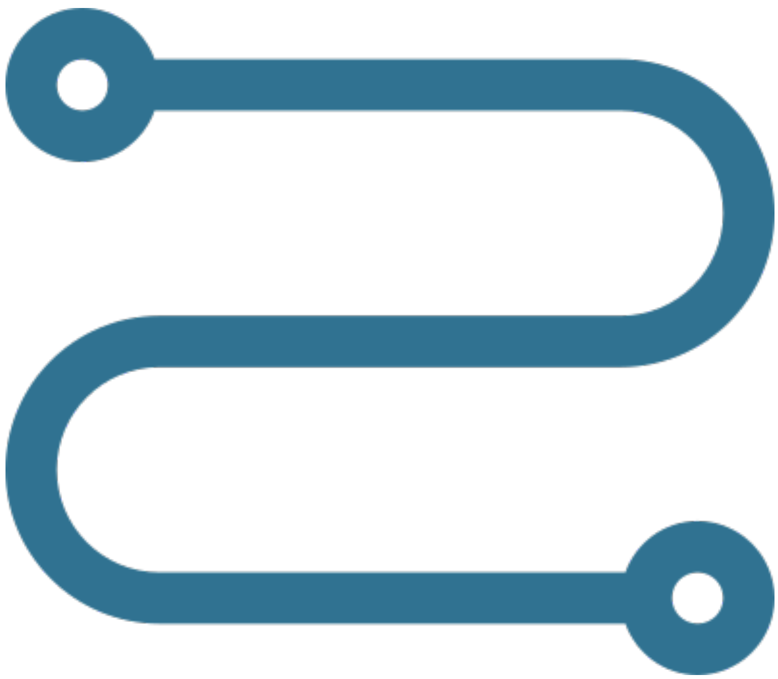
PECL1: VIAJES ENTRE CIUDADES CON RACKET



INTELIGENCIA ARTIFICIAL

Nombre y apellidos

Alvaro de las Heras Fernández

9/4/2019

1.Planteamiento del problema en *Racket*

El problema que se plantea es el de llegar de un origen a un destino dados en un **mapa** representado por un **grafo** con **nodos** y **aristas con peso**. Esto se resolverá mediante **algoritmos de búsqueda** en grafos, empleando el lenguaje de programación *Racket*. El quid de la cuestión es comprobar cuál de estos algoritmos devolverá la mejor **solución**, es decir la **óptima**, siendo esta **completa** con el **menor coste computacional** tanto de **tiempo** como **espacial**. Para ello se plantean varias búsquedas con sus características que son de interés para decidir cuál será la mejor para este problema de caminos en mapas.

Además de la implementación de la búsqueda hay que **leer** de un **archivo** los **datos** de los nodos y aristas, a las que después se aplican unas **conversiones** para ser más **manejables**. También como mejora se ha añadido la posibilidad de exportar una **visualización del grafo** en formato ***dot*** para ***graphViz***, que se puede visualizar fácilmente en http://viz-js.com.

2.Búsquedas de la solución

Las búsquedas que se han planteado son en profundidad, en anchura, optimal y bidireccional en las que algunas tienen implementación con cerrados, además de abiertos. Todas tienen un funcionamiento similar que se distingue por la forma en la que se incorporan nodos a la lista de abiertos, a excepción de la búsqueda bidireccional que trabaja de forma alterna.

Todos ellos emplean **funciones auxiliares** en común que son imprimir-nodo, imprimir-lista destacando sucesores y expandir. Estos dos últimos son los encargados de **examinar el grafo** y sacar los nodos **sucesores** del **nodo actual**, para ello expandir recorre el grafo buscando el nodo que se quiera desarrollar y una vez lo encuentra llama a sucesores que devolverá la lista de sucesores del nodo. Los que emplean cerrados usan la función está en cerrados para comprobarlo, que recorre la lista para ver si se encuentra el nodo actual. La Fig. 1 muestra el código de estas tres funciones

```
;Obtiene una lista con los sucesores del nodo dado
(define (sucesores lista nodo)
  (cond ;Si es vacia la lista no devuelve nada
    [(empty? lista) empty]
    ;Si no es vacia hace una lista con el nodo sucesor y con el resultado de las
posteriores llamadas recursivas
    [(cons (make-nodo (cons (caar lista) (nodo-camino nodo)) (+ (cadar lista)(nodo-
kilometros nodo))) (sucesores (cdr lista) nodo) )]))
;Recorre el grafo para buscar los sucesores del nodo
(define (expandir grafo nodo)
  (cond ;Si es vacio no devuelve nada
    [(empty? grafo) empty]
    ;Si encuentra el nodo entonces llama a sucesores para sacarlos
    [(equal? (caar grafo) (car (nodo-camino nodo)))(sucesores (cdar grafo) nodo)]
    ;Si no se vuelve a llamar a la funcion con los grupos restantes
    [else  (expandir (cdr grafo) nodo)]))

;Comprueba si el nodo esta en la lista de cerrados devolviendo true o false
(define (esta-nodo-en-cerrados nodo cerrados)
```

```
  (cond
    ;Si esta vacia devuelve false
    [(empty? cerrados) #f]
    ;Si encuentra el elemento devuelve true
    [(equal? (car cerrados) nodo) #t]
    ;Si no la sigue recorriendo hasta encontrar otro elemento o dejarla vacia
    [else (esta-nodo-en-cerrados nodo (cdr cerrados))]))
```

Fig. 1 código de las funciones de expandir nodos y comprobar cerrados.

Búsqueda en profundidad

La búsqueda en profundidad maneja la lista de abiertos añadiendo siempre al principio los hijos, de tal forma que siempre explorará primero los hijos del nodo actual hasta llegar al final. Esto hace que no garantice la completitud al poder crearse bucles infinitos además de tampoco garantizar la optimalidad de la solución pudiendo haber soluciones mejores que no se hayan explorado. Además, tampoco es especialmente eficiente en grafos con peso, porque no tiene en cuenta el peso de las aristas. En este caso al añadirle lista de cerrados se consigue evitar bucles infinitos, sin lista de cerrados sigue presentando bucles, aunque la complejidad espacial aumenta considerablemente.

La Fig. 2 muestra el código del algoritmo con cerrados, que es algo más complejo porque requiere comprobar si se encuentra en cerrados el nodo y de añadirlo en caso contrario, si se eliminan esas condiciones tendríamos la búsqueda sin cerrados.

```
(define (busqueda-en-profundidad-cerrados abiertos cerrados tam-grafo)
   ;Si no esta vacia abiertos continua
  (unless (empty? abiertos)
     ;Definimos actual como el primero de abiertos
    (let ([actual (car abiertos)])
      ;Comprobamos que el nodo actual no haya sido visitado ya en cerrados
      (cond
       ;Si ya ha sido visitados se vuelve a buscar con el siguiente nodo de abiertos
        [(esta-nodo-en-cerrados (car (nodo-camino actual)) cerrados) (busqueda-en-
profundidad-cerrados (cdr abiertos) cerrados tam-grafo)]
        ;Si no se comprueba si es el final mostramos los datos de actual y abiertos
        [else (display "\nActual:\n-------\n")
         (imprimir-camino actual)
         (newline)
         (display "\nAbiertos:\n---------\n")
         (muestra-caminos (cdr abiertos))
         (cond
    ;Si coincide con el destino entonces devolvemos el camino y finaliza la busqueda
           [(equal? ciudad-final (car (nodo-camino actual))) (newline)(display
"Camino final: ")(imprimir-camino actual) actual]
           ;Si la lista de abiertos excede el tamano del grafo entonces finaliza la
ejecucion y devuelve nodo de fallo
           [(>= (length cerrados) (- tam-grafo 1)) (display "\nSe han recorrido
todos los nodos pero no se ha encontrado el camino\n") (make-nodo (list) -1)]
            ;Si no vuelve a realizar la busqueda expandiendo el nodo actual
```

```
          [else (busqueda-en-profundidad-cerrados
;Al ser en profundidad tienen prioridad los hijos del nodo actual en abiertos (LIFO)
                 (append (expandir grafo actual) (cdr abiertos)) (cons (car (nodo-
camino actual)) cerrados) tam-grafo)])]))))
```

Fig. 2 código del algoritmo de búsqueda en profundidad con cerrados.

Búsqueda en anchura

La búsqueda en **anchura** maneja la lista de **abiertos** recorriendo los nodos que se encuentran al mismo nivel, para ello añade los **sucesores** del actual al **final de abiertos**, se podría interpretar como una cola **FIFO**. Este método **garantiza** la **completitud** y **optimalidad** porque se recorren **todos** los nodos de un nivel hasta encontrar la **primera** solución, que es la óptima. Sin embargo, tiene un **gran coste** espacial y temporal porque tiene que recorrer todos los nodos del grafo lo que puede suponer una **complejidad** del orden **exponencial**, especialmente cuando se le añade la lista de **cerrados**, para evitar ver nodos ya vistos. Aplicado a un grafo con **coste no** es especialmente **útil** porque no aprovecha esa información porque simplemente se limitará a recorrerlo como uno sin costes.

En el código que muestra la Fig. 3 se puede ver su implementación **sin cerrados** (también está desarrollado con cerrados). Se observa que no tiene las condiciones de cerrados para añadir o no visitar y que es muy **similar** con el resto de las **búsquedas** cambiando el manejo de abiertos.

```
;Busqueda en anchura
(define (busqueda-en-anchura abiertos)
  ;Si no esta vacia abiertos continua
  (unless (empty? abiertos)
    ;Definimos actual como el primero de abiertos
    (let ([actual (car abiertos)])
      ;Mostramos los datos de actual y abiertos
      (display "\nActual:\n-------\n")
      (imprimir-camino actual)
      (newline)
      (display "\nAbiertos:\n---------\n")
      (muestra-caminos (cdr abiertos))
      (cond
    ;Si coincide con el destino entonces devolvemos el camino y finaliza la busqueda
        [(equal? ciudad-final (car (nodo-camino actual)))(display "Camino final: ")
        (imprimir-camino actual) actual]
        ;Si no vuelve a realizar la busqueda expandiendo el nodo actual
        [else (busqueda-en-anchura
               ;Al ser en anchura tienen prioridad los abiertos mas antiguos (FIFO)
                   (append (cdr abiertos) (expandir grafo actual)))]))))
```

Fig. 3 código del algoritmo de búsqueda en anchura sin cerrados.

Búsqueda optimal

La búsqueda **optimal** o **coste uniforme** requiere de un **grafo** con **aristas** con **peso**, en este caso si se dispone de uno. Esta búsqueda se caracteriza porque la lista de **abiertos** se construye de forma ordenada estando al principio los caminos de menor coste. Por tanto, cuando se obtienen los sucesores del nodo actual se insertan de forma ordenada en abiertos, para después seguir examinando. Al realizar esto se garantiza una búsqueda completa, óptima y que mejora el coste de la búsqueda en anchura, porque se ahorran nodos lejanos que visitar. Se podría ver como un círculo que se va expandiendo en función de la distancia hasta llegar al destino.

En el código que muestra la Fig. 4 se observa como es similar al resto con sus condiciones de cerrados, pero como difiere en el tratamiento de abiertos, que en este caso son insertados en orden mediante la llamada de la función de inserta-ordenados-nodos e inserta-nodo, que insertará cada nodo en orden en abiertos en función del coste acumulado del nodo (distancia).

```
;Inserta ordenado un nodo en la lista
(define (inserta-ordenado nodo lista)
  (cond
    ;Si esta vacia inserta el nodo y devuelve la nueva lista
    [(empty? lista) (list nodo)]
    ;Si los kilometros son menores construye la lista añadiendo primero el nodo
    [(< (nodo-kilometros nodo) (nodo-kilometros (car lista))) (cons nodo lista)]
    ;Si no se añade la cabeza de la lista y se vuelve a llamar a la funcion con la
cola
    [else (cons (car lista) (inserta-ordenado nodo (cdr lista)))]))

;Inserta varios nodos ordenados un en la lista
(define (inserta-ordenados-nodos nodos lista)
  (cond
    ;Si no hay nodos que insertar devuelve la lista
    [(empty? nodos) lista]
    ;Si hay nodos que insertar llama a insertar ordenados pasandole un nodo y repite
el proceso con el siguiente nodo
    [else (inserta-ordenados-nodos (cdr nodos)(inserta-ordenado (car nodos)
lista))]))

;Busqueda optimal con cerrados
(define (busqueda-optimal-cerrados abiertos cerrados tam-grafo)
  ;Si no esta vacia abiertos continua
  (unless (empty? abiertos)
    ;Definimos actual como el primero de abiertos
    (let ([actual (car abiertos)])
      ;Comprobamos que el nodo actual no haya sido visitado ya en cerrados
      (cond
       ;Si ya ha sido visitados se vuelve a buscar con el siguiente nodo de abiertos
        [(esta-nodo-en-cerrados (car (nodo-camino actual)) cerrados) (busqueda-
optimal-cerrados (cdr abiertos) cerrados tam-grafo)]
        ;Si no se comprueba si es el final
        ;Mostramos los datos de actual y abiertos
        [else (display "\nActual:\n-------\n")
         (imprimir-camino actual)(newline)
         (display "\nAbiertos:\n---------\n")
```

```
        (muestra-caminos (cdr abiertos))
        (cond
    ;Si coincide con el destino entonces devolvemos el camino y finaliza la busqueda
          [(equal? ciudad-final (car (nodo-camino actual))) (newline)(display
"Camino final: ")(imprimir-camino actual) actual]
            ;Si la lista de abiertos excede el tamano del grafo entonces finaliza la
ejecucion y devuelve nodo de fallo
          [(>= (length cerrados) (- tam-grafo 1)) (display "\nSe han recorrido
todos los nodos pero no se ha encontrado el camino\n") (make-nodo (list) -1)]
          ;Si no vuelve a realizar la busqueda expandiendo el nodo actual
          [else (busqueda-optimal-cerrados
             ;Al ser optimal tienen prioridad los que tienen menor coste de camino
                  (inserta-ordenados-nodos (expandir grafo actual) (cdr abiertos))
          (cons (car (nodo-camino actual)) cerrados) tam-grafo)])])))
```

Fig. 4 código del algoritmo de búsqueda optimal con cerrados.

Búsqueda bidireccional

La búsqueda bidireccional se caracteriza porque emplea dos algoritmos de búsqueda que van alternando los pasos, estos algoritmos pueden ser los tipos que se quieran, pero en función de eso tendrán más coste o podrán no ser completos u óptimos. Además, requieren que sea posible ir del origen al destino y del destino al origen. En este caso se ha optado por coger dos búsquedas de coste uniforme, para aprovechar así la información que hay sobre los caminos y reducir costes en comparación a otras búsquedas. Los costes se reducen en problemas con bastante ramificación porque aplicando una única búsqueda la cantidad de nodos crece exponencialmente como se ve en la Fig. 5, en el círculo más grande que se corresponde a una única búsqueda optimal, mientras que con la bidireccional este coste exponencial se reduce por dos como la suma de dos áreas con una ramificación menor. Por eso esta es la mejor búsqueda para este problema porque se adapta a los requisitos, ofreciendo una solución completa, óptima y con un coste mucho menor. La única pega se encuentra en la comprobación del nodo en común entre ambas listas de abiertos, que se requiere de hacer en cada turno, pero que quizá aplicando heurística podría verse reducido más.

En el código que muestra la Fig. 6 se puede ver su implementación sin cerrados. Esta búsqueda difiere del resto al emplear dos búsquedas que se alternan y comienzan de los nodos inicial y final. En este caso habrá dos listas de abiertos que se comprobarán mediante unas funciones auxiliares hasta que se encuentre un nodo en común en ambas, una vez ocurre eso acaba.

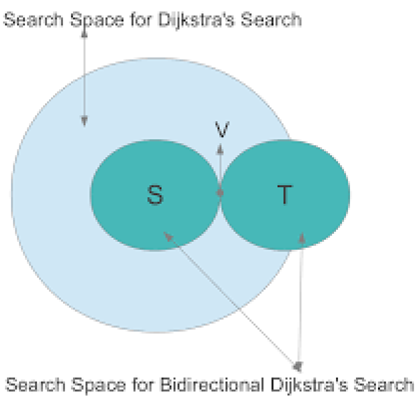


Fig. 5 Comparación entre bidireccional (de dos optimales) y optimal.

```
      ;Busqueda bidireccional

(define (busqueda-bidireccional abiertos-origen abiertos-destino)
      ;Solo usaremos la rama cuando no este vacia
    (unless (empty? abiertos-origen)
    ;definimos el nodo actual como el primero de abiertos
    (let ([actual-origen (car abiertos-origen)])
      ;Mostraos el actual y abiertos
      (display "\nActual desde origen:\n-------\n")
      (imprimir-camino actual-origen)
      (newline)
      (display "\nAbiertos desde origen:\n---------\n")
      (muestra-caminos (cdr abiertos-origen))
      (unless (empty? abiertos-destino)
          ;definimos el nodo actual como el primero de abiertos
          (let ([actual-destino (car abiertos-destino)])
          ;Mostraos el actual y abiertos
          (display "\nActual desde destino:\n-------\n")
          (imprimir-camino actual-destino)
          (newline)
          (display "\nAbiertos desde destino:\n---------\n")
          (muestra-caminos (cdr abiertos-destino))
          ;Comprobamos si es el nodo final o expandimos con un nuevo nodo
          (cond
            [(not (empty? (comprobar-caminos-en-caminos abiertos-origen
       abiertos-destino))) (display "Camino final: ") (imprimir-camino
       (car (comprobar-caminos-en-caminos abiertos-origen abiertos-destino)))
       (car (comprobar-caminos-en-caminos abiertos-origen abiertos-destino))]
      [else (busqueda-bidireccional (inserta-ordenados-nodos (expandir grafo
      actual- origen) (cdr abiertos-origen))
      (inserta-ordenados-nodos (expandir grafo actual-destino)
      (cdr abiertos-destino)))])))))))
```

Fig. 6 código del algoritmo de búsqueda bidireccional sin cerrados.

3.Leer datos del fichero

Para leer los datos del fichero se emplea una función para leer los datos dada por el profesor, pero para poder trabajar mejor con los datos se realiza una conversión agrupando los datos por un nodo inicial en común. Para leer se emplea el método read-line hasta llegar al EOF (*End Of File*), cada línea genera una lista, en caso de que no esté vacía, con la arista y coste. Para la reducción a un solo nodo comprobamos primero si hay duplicados en ese caso construimos una nueva lista con esos dejando únicamente el nodo inicial, después se eliminan estos de la lista. Para finalmente tener el grafo con el formato reducido que ahorra visitar una gran cantidad de nodos.

```
#lang racket
;Importamos la biblioteca graph para la representacion visual
(require graph)
```

```
;------------------------------- LEER GRAFO DEL FICHERO --------------------

;Crea cada linea del grafo
(define (crea-grafo line)
  ;La lista seran los datos separados de la linea
  (let ([lista (string-split line)])
    (cond
      ;Si la linea es vacia devuelve la linea vacia
      [(empty? lista)'()]
      ;Si no construye la lista
      [else (list (car lista) ( list (cadr lista) (string->number (caddr lista))))])))

;Lee el grafo linea a linea
(define (read-graph file)
  ;Definimos linea como leer linea del archivo
  (let ([line (read-line file 'any)])
    ;Comprueba que no es el final del fichero
    (if (eof-object? line)
        empty
        ;Construye el grafo con los datos
        (cons (crea-grafo line) (read-graph file)))))

;Adapta el formato leido al formato de trabajo
(define (reducir-formato grafo)
   (cond
     ;Si esta vacio no devuelve nada
    [(empty? grafo) grafo]
    ;Si existen duplicados los obtiene y los elimina para dar el formato deseado
    [(not (empty?(comprobar-duplicados (caar grafo)(cdr grafo)))) (cons (cons (caar
grafo) (comprobar-duplicados (caar grafo) grafo)) (reducir-formato (eliminar-duplicados
(caar grafo) grafo)))]
    ;Si no se sigue construyendo el grafo con el nodo dado
    [else (cons (car grafo) (reducir-formato (cdr grafo)))]))

;Comprueba si hay nodos con la ciudad de origen en comun y los fusiona en una lista
(define (comprobar-duplicados nodo grafo)
  (cond
    ;Si esta vacio no devuelve nada
    [(empty? grafo) empty]
    ;Si el nodo coincide con lo que buscamos se anade
    [(equal? nodo (caar grafo)) (append (cdar grafo) (comprobar-duplicados nodo (cdr
grafo)))]
    ;Si no se vuelve a ejecutar para encontrar otras coincidencias
    [else (comprobar-duplicados nodo (cdr grafo))]))

;Elimina los nodos del grafo que tengan el mismo nodo inicial que se pasa por parametro
(define (eliminar-duplicados nodo grafo)
  (cond
```

```
    ;Si esta vacio devuelve la lista vacia
    [(empty? grafo) (list)]
    ;Si el nodo que estamos buscando se repite entonces se salta
    [(equal? nodo (caar grafo)) (eliminar-duplicados nodo (cdr grafo))]
    ;Si no es el nodo que se busca entonces anade al grafo el valor
    [else (cons (car grafo) (eliminar-duplicados nodo (cdr grafo)))]))

;Definimos el grafo que se va a emplear leyendolo del archivo
(define grafo (reducir-formato (call-with-input-file "entrada.txt" read-graph)))
```

Fig. 7 código del algoritmo de búsqueda optimal con cerrados.

4.Implementación gráfica

La implementación gráfica ha sido una mejora añadida para así poder examinar los resultados en un grafo visual. Para ello se emplea la librería *graph* de Racket que permite generar grafos en formato *dot*, que son interpretables por la herramienta ***graphViz***, o su versión on-line http://viz-js.com.

En este caso se ha realizado una conversión del grafo a un modelo admisible por la librería *graph*, después para mostrar el camino se ha coloreado el grafo y modificado los nodos que estaban en el camino para representarlo. En la Fig. 8 se puede observar como se crea el grafo en ***graphViz*** y como se resalta el camino.

```
;Creacion del grafo y coloracion para despues obtener su graphviz
(define g (weighted-graph/undirected (convertir-grafo grafo)))
(define dot (graphviz g #:colors (coloring/brelaz g)))
;Funcion que permite resaltar el camino en graphViz
(define (mostrar-camino-grafico dot camino)
  (cond
    ;Si esta vacio el camino a resaltar devuelve el dot
    [(empty? camino) dot]
    ;Si no esta vacio el camino se busca el nodo de camino y se resalta
    [else (mostrar-camino-grafico (string-replace dot (string-append "label=\"" (car
camino) "\"") (string-append "label=\"" (car camino) "\", style=filled")) (cdr
camino))])
  )
```

Fig. 8 Generación del grafo en *graphViz* y marcado del camino.

Se puede ver la representación en la Fig. 9 del mapa que se da con las distancias y el camino resaltado para ir de Vigo a Cádiz mediante búsqueda optimal.

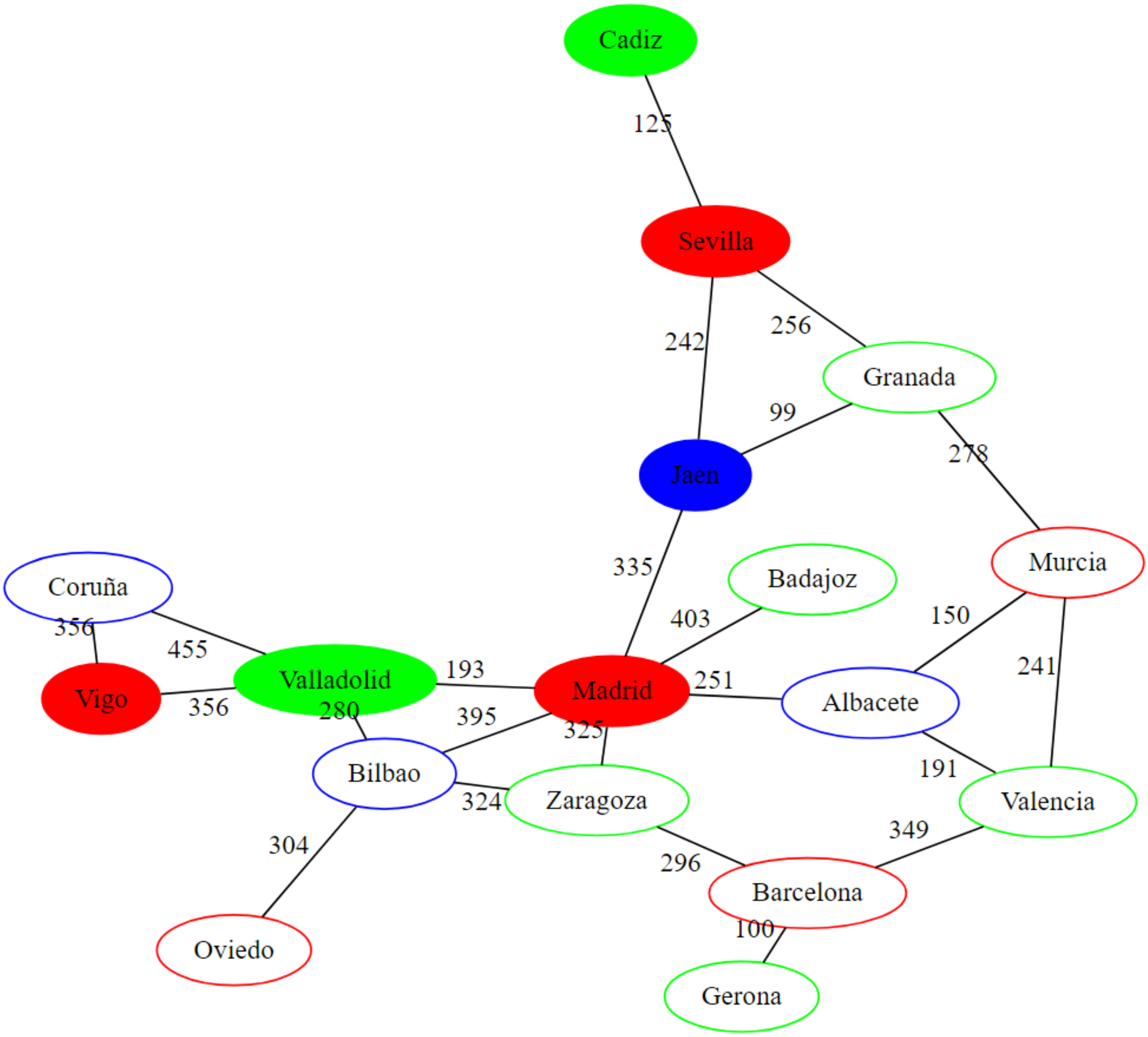


Fig. 9 Grafo de ciudades con el camino más corto entre Vigo y Cádiz.

5.Conclusion

En conclusión, a la realización de la práctica se pueden sacar las ideas de la **fácil modelización** de los problemas de **carreteras** como un **grafo** con **nodos** y **aristas** que simulan las **ciudades** y **carreteras** que las conectan, de esta forma es fácil **computabilizarlo** y aplicar **algoritmos** sobre éste. Estos algoritmos pueden ser como los que se han trabajado de **búsqueda**, para obtener el **camino** que **une dos ciudades** pudiendo ser este camino **óptimo** y **completo** según el algoritmo utilizado, con un cierto **coste espacial** y **temporal**. Entre los algoritmos empleados destaca el **bidireccional** con **dos búsquedas de coste uniforme**, porque en este caso se puede ir del destino al origen y viceversa ahorrando recorrer gran cantidad de nodos. Además de aprovechar que se conoce el coste entre ciudad y ciudad, eligiendo así la ruta más corta.

También se ha visto el potencial de *Racket* al poder **leer ficheros** y la posibilidad de emplear *graphViz* para poder **representar** estos **grafos**.