MEMORIA TP6 -1
Búsqueda local y Propagación de restricciones

Inteligencia Artificial

Curso 2019-2020

Autor

Fernando Peña Bes

nia: 756012

Grado en Ingeniería Informática

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

10 de noviembre de 2019

Índice general

Introducción

En este documento se trata el trabajo realizado en el trabajo práctico TP6 de la asignatura Inteligencia Artificial de la Universidad de Zaragoza.

El trabajo consta de tres tareas. En la primera se resuelven problemas de **búsqueda local** y optimización y aplicándolo al problema de las **8-reinas**. En la segunda, se resuelve el problema de resolución de **sudokus** mediante **propagación de restricciones**, y por último, en la tercera se combina la idea de búsqueda local con la propagación de restricciones usando el algoritmo `min-conflicts`, aplicándolo también al problema de las 8-reinas.

Para realizar la práctica se ha usado el entorno de desarrollo Eclipse para Java y el código perteneciente al libro *Artificial Intelligence: A Modern Approach* [1], en la versión 1.8 (se puede descargar en `https://github.com/aimacode/aima-java/releases/tag/aima3e-v1.8.1`).

Tarea 1

Resolución del problema de las 8 reinas con búsqueda local

1.1. Objetivo

El trabajo de esta tarea consiste en la familizarización con los algoritmos de búsqueda local **Hill-Climbing**, **Simulated Annealing** y **Genetic Algorithm**, utilizando el problema de las 8-reinas. Se utiliza el código del paquete `aima.core.search.local`, donde se implementan los algoritmos anteriores, y se parte del código de la clase `NQueensDemo` del paquete `aima.core.search.local`, en el que se muestra la resolución del problema usando diferentes estrategias de búsqueda.

El trabajo realizado en esta tarea se encuentra en el fichero `NQueensLocal.java` dentro del paquete `aima.gui.demo.search`. En el método `main`, se ponen en ejecución los métodos implementados.

1.2. Hill Climbing Search (Escalada)

Este algoritmo parte de un cierto estado inicial, en cada paso genera nodos correspondientes a los estados vecinos y expande el que tenga un valor que maximice o minimice la función objetivo (en el caso de que haya empate, lo resuelve de manera aleatoria). El algoritmo para en el momento se alcanza un "pico" sin vecinos con mejor valor que el estado actual, no mira más allá de los vecinos inmediatos. El algoritmo encuentra máximos (o mínimos) locales.

En el caso del problema de las 8-reinas, se parte de un tablero inicial de ajedrez 8×8 con una reina colocada en cada columna. La función sucesor consiste en mover una única reina a otro cuadrado en la misma columna. La función objetivo (o de coste heurístico) es el número de pares de reinas que se atacan entre sí, por lo que se intenta minimizar esta función. La búsqueda Hill-Climbing para cuando ninguno de los sucesores de un estado tiene un menor número de pares de reinas que se ataquen entre ellas. El problema se resuelve cuando la función de evaluación vale 0, por lo que el algoritmo tiene que encontrar un mínimo global. El algoritmo sólo podrá dar una solución cuando el mínimo local encontrado coincida con uno de los globales.

Implementación

Se ha implementado el método `nQueensHillClimbingSearch_Statistics(numExperiments)`, que realiza `numExperiments` veces la búsqueda Hill-Climbing y muestra el porcentaje de éxitos y fallos, media de pasos al fallar y media de pasos en éxito. Este método llama a `nQueensHillClimbingSearch()` para realizar cada búsqueda.

Se ha implementado, además, un método llamado `generateUnusedRandomBoard()`, que devuelve un tablero con una reina colocada aleatoriamente en cada una de las columnas. Además, se asegura que ese tablero no esté ya dentro del vector `tablerosUsados`, de forma que se pueda evitar generar tableros repetidos. Cada nuevo tablero creado se añade a `tablerosUsados`. El método que invoque a este método tendrá que ser quien se encargue de vaciar el vector si es necesario. Este método se apoya en otro creado dentro de la clase `NQueensBoard` del paquete `aima.core.enviroment` llamado `putNRandomQueens()` que es el que accede al atributo que almacena el tablero y lo modifica poniendo las N reinas de forma aleatoria.

Con 8 reinas, el número máximo de tableros que se pueden generar con una reina en cada columna es $8! = 40320$, y el número total de soluciones al problema es 92 soluciones distintas (12 juntando simétricas).

Pruebas y resultados

Para probar el algoritmo Hill-Climbing, se han realizado 10 000 experimentos a partir de tableros aleatorios iniciales de las 8-reinas no repetidos.

El resultado de la ejecución se puede ver en la tabla 1.1.

```
NQueens HillClimbing con 10000 estados iniciales diferentes  -->
Fallos: 84,40
Coste medio fallos: 3,25
Exitos: 15,60
Coste medio exitos: 4,18
```

Tabla 1.1: Resultados de ejecución `nQueensHillClimbingSearch(10000)`

El espacio de estados del problema es $8^8 \approx 17\,000\,000$. Según la teoría de la asignatura, el algoritmo de escalada debería atascarse el 86 % de las veces, dar 3 pasos en los fallos y 4 en los aciertos. En las pruebas realizadas podemos ver que el algoritmo de escalada falla en el 84,40 % de las casos y resuelve el problema el 15,60 % de las veces, cuando se atasca, realiza 3,25 pasos de media y cuando acierta 4,18. Los resultados son muy similares a los teóricos.

Este algoritmo falla en la mayoría de ocasiones, aunque podemos ver que cuando se atasca sale rápidamente, lo que no está mal para un problema con un espacio de estados tan grande.

1.3. Random Restart Hill Climbing Search

Una solución que intenta mejorar el rendimiento de Hill-Climbing es el algoritmo **Random-Restart Hill-Climbing**. Este algoritmo intenta evitar atascarse en un máximo local. Ejecuta el algoritmo Hill-Climbing y si se atasca, vuelve a ejecutarlo sobre un nuevo tablero aleatorio. La búsqueda es completa, ya que, si se realizan infinitos intentos, la probabilidad de generar el estado objetivo es 1.

Implementación

Se ha implementado el método `nQueensRandomRestartHillClimbing()` que resuelve el problema de las 8 Reinas aplicando el algoritmo de Random-Restart Hill-Climbing y muestra estadísticas del número de intentos (número de cambios de tablero), número de fallos, coste medio de fallos, coste del éxito y coste medio del éxito. Los nuevos tableros se generan con la función `generateUnusedRandomBoard()`, para no repetir la búsqueda sobre un tablero ya usado.

Pruebas y resultados

En la tabla 1.2 se muestran los resultados de 4 ejecuciones diferentes de la función.

NQueens RestartHillClimbing --> Search Outcome=SOLUTION_FOUND Final State= -Q------ ------Q- --Q----- -----Q-- -------Q ----Q--- Q------- ---Q---- Numero de intentos: 6 Fallos: 5 Coste medio fallos: 3,20 Coste exito: 4 Coste medio exito: 5,00	Queens RestartHillClimbing -->\nSearch Outcome=SOLUTION_FOUND\nFinal State=\n----Q---\nQ-------\n-------Q\n-----Q--\n--Q-----\n------Q-\n-Q------\n---Q----\n\nNumero de intentos: 5\nFallos: 4\nCoste medio fallos: 2,75\nCoste exito: 5\nCoste medio exito: 6,00
NQueens RestartHillClimbing -->\nSearch Outcome=SOLUTION_FOUND\nFinal State=\n---Q----\n-Q------\n-------Q\n-----Q--\nQ-------\n--Q-----\n----Q---\n------Q-\n\nNumero de intentos: 10\nFallos: 9\nCoste medio fallos: 2,89\nCoste exito: 4\nCoste medio exito: 5,00	NQueens RestartHillClimbing -->\nSearch Outcome=SOLUTION_FOUND\nFinal State=\n-Q------\n----Q---\n------Q-\nQ-------\n--Q-----\n-------Q\n-----Q--\n---Q----\n\nNumero de intentos: 1\nFallos: 0\nCoste medio fallos: NaN\nCoste exito: 5\nCoste medio exito: 6,00

Tabla 1.2: Resultados de ejecución `nQueensHillClimbingSearch(10000)`

Como hemos visto experimentalmente en la sección anterior, el algoritmo Hill-Climbing tiene una probabilidad de éxito $p \approx 0{,}14$ para este problema. El número de reintentos es $1/p$ así que, de media, se requerirán $1/0{,}14 = 7$ iteraciones para tener éxito (6 fallos y 1 éxito). También hemos comprobado que el coste en los fallos era 3 pasos y en los aciertos 4 (redondeando), por tanto, el algoritmo resolverá el problema en $3 \times 6 + 4 \times 1 = 22$ pasos de media.

En la tabla podemos ver como las estadísticas de los ejemplos se ajustan a este valor.

Estos resultados se corroboraron también de manera experimental, usando el método `nQueenRestartHillClimbingRestart_Test(int numExperiment)`. Se obtuvo una media de número de reintentos de 6,32 y un coste medio de 21,53 pasos para 10 000 ejecuciones. Conforme el número de experimentos tendiera a infinito, el resultado estaría más cerca al calculado.

1.4. Simulated Annealing (Enfriamiento Simulado)

El algoritmo Simulated Annealing (SA) genera un estado aleatorio sucesor. Si el coste del nuevo estado es mejor, el cambio se acepta. Por el contrario, si se produce un incremento en la función de evaluación, el cambio será aceptado con una cierta probabilidad. El SA acepta con mayor probabilidad estados peores, pero según se va "enfriando", la probabilidad de aceptar estados peores es menor.

El algoritmo se controla con los parámetros T_{final}, k y δ. En el algoritmo, una variable T controla el paso de iteración en la ejecución, el algoritmo termina cuando $T = T_{final}$. El valor de T_final es necesario determinarlo experimentalmente. k determina la velocidad que tarda en comenzar a decrecer la temperatura y δ mide lo rápido que desciende la temperatura. Conforme se aumenta el valor de k se aleja el punto en el que se anula la probabilidad de aceptar peores estados y al disminuir el valor de δ se alarga el tiempo necesario para converger. Fijados unos valores (k, δ) el número de iteraciones mínimos que necesitamos para T, será el de aquella iteración en que la P(aceptación) se anule. A partir de esta iteración, el algoritmo sólo aceptará mejores soluciones.

Implementación

Se ha implementado un método llamado `nQueensSimulatedAnnealing_Statistics(int numExperiments)` que realiza `numExperiments` experimentos con el algoritmo Simulated Annealing partiendo de estados iniciales aleatorios no repetidos y muestra el porcentaje de éxitos, fallos, media de pasos al fallar y media de pasos en éxito. Este método hace una llamada a `nQueensSimulatedAnnealingSearch()` cada vez que se necesita realizar una ejecución del algoritmo. Se han definido las constantes de clase `k`, `lam` y `limit` que permiten establecer los parámetros de ejecución del algoritmo.

Pruebas y resultados

Se han realizado 1 000 experimentos con el algoritmo implementado y en la tabla 1.3 se muestran los resultados obtenidos.

```
NQueens HillClimbing con 10000 estados iniciales diferentes  -->
Fallos: 84,40
Coste medio fallos: 4,25
Exitos: 15,60
Coste medio exitos: 5,18
```

Tabla 1.3: Resultados de ejecución `nQueensHillClimbingSearch(10000)`

Poner pruebas y gráficas para ver que valores se adaptan mejor al problema

1.5. Simulated Annealing Restart

El algoritmo Simulated Annealing Restart funciona de forma similar al Random-Restart Hill-Climbing, reinicia a un estado inicial aleatorio cuando falla hasta que obtiene éxito en las búsquedas.

Implementación

Se ha implementado una clase llamada `nQueensSimulatedAnnealingRestart()` que ejecuta el algoritmo y muestra el número de reintentos, la solución, el número de fallos y el coste del éxito.

Pruebas y resultados

En la tabla 1.4 se muestran 4 resultados de ejecución del algoritmo para los mejores valores iniciales anteriores:

```
NQueens RestartHillClimbing -->
Search Outcome=SOLUTION_FOUND
Final State=
-Q------
------Q-
--Q-----
-----Q--
-------Q
----Q---
Q-------
---Q----

Numero de intentos: 6
Fallos: 5
Coste medio fallos: 4,20
Coste exito: 5
Coste medio exito: 5,00
```

```
Queens RestartHillClimbing -->
Search Outcome=SOLUTION_FOUND
Final State=
----Q---
Q-------
-------Q
-----Q--
--Q-----
------Q-
-Q------
---Q----

Numero de intentos: 5
Fallos: 4
Coste medio fallos: 3,75
Coste exito: 6
Coste medio exito: 6,00
```

```
NQueens RestartHillClimbing -->
Search Outcome=SOLUTION_FOUND
Final State=
---Q----
-Q------
-------Q
-----Q--
Q-------
--Q-----
----Q---
------Q-

Numero de intentos: 10
Fallos: 9
Coste medio fallos: 3,89
Coste exito: 5
Coste medio exito: 5,00
```

```
NQueens RestartHillClimbing -->
Search Outcome=SOLUTION_FOUND
Final State=
-Q------
----Q---
------Q-
Q-------
--Q-----
-------Q
-----Q--
---Q----

Numero de intentos: 1
Fallos: 0
Coste medio fallos: NaN
Coste exito: 6
Coste medio exito: 6,00
```

Tabla 1.4: Resultados de ejecución `nQueensHillClimbingSearch(10000)`

1.6. Genetic Algorithm (Algoritmos Genéticos)

Los algoritmos genéticos son una variación de la búsqueda en haz estocástica, donde los estados sucesores se generan por la combinación de dos estaos padres en vez de modificando un sólo estado. Estos algoritmos empiezan con una población de k estados aleatoriamente generados. Cada estados es evaluado por una función de fitness, que devuelve mayor valor cuanto más cerca esté el estado de la solución. En cada iteración se selecciona una pareja de estados de forma aleatoria, aunque los mejores estados tendrán más probabilidad de ser elegidos. Estos estados se unen de forma aleatoria

formando un hijo. Además, existe la posibilidad de mutación el hijo, de forma que cambie su estado de alguna forma. Este proceso se repite hasta que la función de evaluación evalúe a un estado y obtenga el valor objetivo.

En el ejemplo de las 8-reinas, la función de fitness devuelve el número de reinas que no se atacan. El valor objetivo es 28, ya que corresponde con el número total de parejas con 8 reinas donde el orden no importa (combinaciones sin repetición) es:

$$\binom{8}{2} = \frac{8!}{2!(8-2)!} = 28$$

Cada tablero con una cadena de 8 dígitos, donde cada uno representa la posición de la reina en cada columna. Al unir dos tableros, cada cadena se divide en un punto aleatorio y se juntan. Cuando los dos padres son diferentes, el hijo puede ser todavía más diferente. A menudo, la población inicial es muy diversa por lo que al principio se dan grandes pasos en el espacio de estados, pero confome se avanza en la búsqueda, los estados van convergiendo (como la búsqueda Simulated Annealing). Las mutaciones son interesantes porque evitan converger hacia estados lejanos del objetivo. Este algoritmo puede combinar estados cercanos a la solución que han ido evolucionado de forma separada, lo que consigue elevar el nivel de granularidad de la búsqueda. La idea es que si un componente es bueno, seguramente lo siga siendo combinado con otros.

Implementación

Se ha implementado el método `nQueensGeneticAlgorithmSearch()` que realiza la búsqueda con el algoritmo genético. Los parámetros iniciales se pueden establecer cambiando los valores de las constantes de clase `sizePoblacion` y `probMutacion`. Una vez realizada la ejecución, el algoritmo muestra el mejor individuo encontrado, su fitness, el número de iteraciones y el tiempo de ejecución total.

Pruebas y resultados

Poner pruebas diferentes y gráficas para establecer los mejores parámetros iniciales.

1.7. Conclusiones

En esta práctica se han evaluado diferentes algoritmos y heurísticas y se ha visto cuales son más adecuadas para resolver el problema del 8-puzzle. El mejor algoritmo de los evaluados, es el A* con la heurística de Manhattan.

Tarea 2

Resolución de sudokus mediante propagación de restricciones y búsqueda

Tarea 3

Propagación de restricciones y búsqueda local

Conclusiones

Bibliografía

[1] Stuart Rusell and Peter Norvig, *Artificial Intelligence: A Modern Approach*.

[2] *Apuntes de la asignatura Inteligencia Artificial*, Curso 2019-20.