Tarea 4 - Optimización estocástica

Erick Salvador Alvarez Valencia

CIMAT A.C.,
erick.alvarez@cimat.mx

Resumen En el presente reporte se hablará sobre la implementación de un algoritmo de programación dinámica que fue usado como búsqueda local en el problema del sudoku. De la misma forma se hablará sobre los resultados obtenidos con instancias de 9x9 y de 16x16 así como una pequeña comparación con el algoritmo de búsqueda local implementado en la tarea pasada.

Keywords: Sudoku, búsqueda local, programación dinámica.

1. Correcciones y adaptaciones con respecto al programa de la tarea anterior

Para comenzar, lo primero que se realizó fue modificar el código que ya se tenía con respecto a dos cosas: La primera, hacer una pequeña adaptación a la búsqueda local ya programada. Lo que se hizo fue generar todos los vecinos de un estado, luego permutarlos aleatoriamente y finalmente ver cuál de ellos generaba un mejor fitness y con el cual se comenzaba la búsqueda desde el principio pero de ese nuevo estado. Anteriormente lo que se hacía era elegir bloques de mamnera aleatoria y posteriormente generar los vecinos de el bloque elegido y aplicar los cambios en el orden que se iban generando estos vecinos lo cual no generaba una búsqueda lo suficientemente uniforme. Al hacer este cambio se notó que la búsqueda generaba mejores resultados que la versión anterior.
Otro cambio importante que se hizo al programa fue adaptarlo para que pudiera procesar sudokus de tamaño nxn donde n tiene raíz cuadrada exacta. De esta forma se pudieron procesar sudokus de 16x16 en esta tarea usando las búsquedas locales.

2. Implementación de una búsqueda local óptima

Para esta tarea el enfoque principal fue el de implementar una búsqueda local que fuera óptima para instancias más grandes de un sudoku, ya que la búsqueda local implementada hasta el presente momento generaba todas las permutaciones posibles de números disponibles en cada bloque del sudoku, teniendo tiempo de $O(n!)$ donde n es la cantidad de dígitos disponibles para colocar en ese bloque.
La búsqueda local actual se basa en la técnica de programación dinámica, en

donde el algoritmo se enfoca principalmente en buscar el coste mínimo que se puede lograr con un cierto conjunto de números en un cierto bloque, de igual manera se puede encontrar la permutación de ese conjunto que generó el menor costo.
La idea general es crear una permutación aleatoria de los n bloques que tiene el sudoku y ejecutar la DP a cada uno de estos bloques, luego se verifica si el fitness general del sudoku disminuyó al aplicar la búsqueda a todos sus bloques y de ser así, se repite todo este proceso desde el inicio, pero usando la nueva instancia del sudoku generada por las búsquedas. De caso contrario que no se haya encontrado una mejora, se asume que se ha llegado a un óptimo local, esto porque la DP siempre intentará buscar permutaciones que generen un menor costo, y en el peor de los casos se regresará una permutación que genere el costo que ya se tenía (no hay mejoras).
Para poder implementar la DP es prescindible una tabla de costos local, en la cual constituimos las filas como los números de la permutación para el bloque y las columnas como el número de casilla donde se colocará el número, ahora al consultar la tabla, esta nos dirá el costo asociado de colocar el i-ésimo número de la permutación en la j-ésima casilla disponible. Esta tabla se debe calcular cada vez que se ejecute el algoritmo de DP ya que una modificación a cualquier bloque del sudoku puede afectar los costos que generarían el poner los números en las casillas del bloque actual.
A continuación se muestra el algoritmo de creación de la tabla de costos.

Algorithm 1 Generación de la tabla de costos.

```
1: procedure CostTable(sudoku, gridId, n)
2:     gs ← sqrt(n).
3:     res ← Matrix(n + 1, n + 1).
4:     pinit ← getTablePos(gridId, n).
5:     cId ← 1.
6:     for i = pinit.y → pinit.y + gs do
7:         for j = pinit.x → pinit.x + gs do
8:             if Casilla disponible then
9:                 for k en permutaciones do
10:                    cost ← 0.
11:                    d ← perm_k                ▷ Obtenemos la k-ésima permutación.
12:                    for x = 0 → n - 1 do
13:                        if x ≠ j then
14:                            if sudoku[i][x] = d then      ▷ Si el dígito d coincide con
    alguno ya puesto.
15:                                if sudoku[i][x] no es dígito por defecto then
16:                                    cost ← cost + getCost(i, x).
17:                    for y = 0 → n - 1 do
18:                        if y ≠ i then
19:                            if sudoku[y][j] = d then      ▷ Si el dígito d coincide con
    alguno ya puesto.
20:                                if sudoku[y][j] no es dígito por defecto then
21:                                    cost ← cost + getCost(y, j).
22:                    res[d][cId] ← cost.
23:                cId ← cId + 1.
24:    return res
```

Podemos ver que en el algoritmo anterior iteramos sobre todas las celdas disponibles del bloque actual, al determinar que en efecto la celda está disponible iteramos con todos los posibles números que podemos poner en esa casilla, y por cada número que podemos poner verificamos todos los números de la fila y columna relacionados con esa casilla para ver si hay coincidencias con números repetidos, en caso de haberlas tenemos que aumentar el costo correspondiente de poner ese número en la casilla y para ello llamamos a la función *getCost* la cual añade un costo dependiento si el número con el que se coincide es uno que ya estaba puesto por defecto o no. Al finalizar el algoritmo tendremos la tabla de costo local correspondiente al bloque indicado. Se puede notar que este algoritmo es costoso de ejecutar pero como el tamaño de los bloques no es tan grande no hay tanto problema por ese aspecto.

Ahora en la parte de la DP se tiene el siguiente problema: Hay que colocar un número de la permutación en la casilla i de tal forma que al colocar los demás números en las casillas subsiguientes se nos genere el menor costo posible. De esta forma se puede ver que el problema tiene la propiedad de subestructura óptima ya que podemos generar un problema más pequeño de uno ya definido,

el cual es: Colocar un número en la casilla i y que sea parte de la permutación que genera costo mínimo.
Hay que tener en cuenta que este algoritmo busca en todas las permutaciones posibles hasta encontrar la que de el menor costo a la solución, la ventaja es que más de una vez estaremos llegando a ciertas soluciones que ya habíamos visitado previamente pero como sabemos qué costo generan no tendremos que recalcularlas.
Para poder hacer uso de este algoritmo debemos disponer de una forma de saber qué números hemos usado ya y para ello hay varias formas de tener esto calculado, una de ellas es usar un vector booleano indicando si el i-ésimo número del conjunto ya fue usado, pero esto se puede hacer más eficiente aún, si usamos la representación binaria de un entero para hacer lo que definimos con el vector booleano el cálculo va a ser más rápido ya que las operaciones a nivel de bits se ejecutan directamente en el procesador, además de que vamos a ahorrar un poco de memoría que gastaría el vector booleano. En general se necesitaría un entero de 32 bits para hacer esto anterior y como a lo mucho usaremos 9 bits en los sudokus pequeños y 16 en los grandes no tendremos problemas de memoria en ese aspecto.
A continuación se mostrará el algoritmo usado para generar la búsqueda local en un bloque del sudoku mediante la técnica de programación dinámica.

Algorithm 2 Búsqueda local usando DP.

```
1: procedure DPLocalSearch(sudoku, gridId)
2:     ss ← getSudokuSize(sudoku).
3:     N ← getPermutationSize(sudoku, gridId).          ▷ Obtenemos la cantidad de
   números a colocar en el bloque.
4:     cost ← CostTable(sudoku, gridId, ss).
5:     dp ← Matrix(N + 1, 2^N + 1).
6:     idx ← MatrixOfPairs(N + 1, 2^N + 1).      ▷ Aquí guardaremos la solución que
   vaya generando el menor costo posible
7:     fillMatrix(dp, ∞).          ▷ Llenamos la tabla de DP con valores muy grandes.
8:     dp[0][0] ← 0.
9:     for i = 1 → N do
10:        for j = 0 → 2^N − 1 do
11:            if dp[i][j] < ∞ then
12:                for k = 0 → N do
13:                    if j & (1 << k) = 0 then  ▷ Se verifica si el k-ésimo bit no está
   en uso.
14:                        g ← perm_k.                ▷ Obtenemos la k-ésima permutación.
15:                        if dp[i - 1][j] + cost[d][i] < dp[i][j | (1 << k)] then
16:                            dp[i][j | (1 << k)] ← dp[i - 1][j] + cost[d][i].
17:                            idx[i][j | (1 << k)] ← makePair(j, k).
                    ▷ El costo de la solución óptima se encuentra en dp[N][2^N − 1].
18:    rep ← Vector(N).
19:    res ← Vector(N).      ▷ Creamos dos vectores de tamaño N para encontrar la
   representación.
20:    bfr ← 2^N − 1.
21:    for i = N → 1 do
22:        rep.add(idx[i][bfr].second).
23:        bfr ← idx[i][bfr].first.
24:    reverse(rep).
25:    for i = 0 → rep.size() - 1 do
26:        res.add(Número de ppermutación con índice rep[i]).
27:    return res
```

El algoritmo 2 representa una búsqueda local usando la técnica de programación dinámica como *Bottom-Up* la cual es una versión iterativa del clásico backtracking con memorización. Lo primero que hacemos es calcular la tabla de costo local asociada al bloque, luego creamos la tabla de la DP de tamaño $(N+1)$x2^N la cual representa: La cantidad de casillas en donde hemos puesto un número y la cantidad de números usados mediante una representación binaria. La idea es iterar sobre la cantidad de casillas disponibles y posteriormente sobre todas las representaciones binarias de esos N números y verificamos si ya hemos calculado la solución asociada a los índices i, j de ser así iteramos sobre todos los números disponibles de la permutación y verificamos si podemos colocar el k-ésimo número, esto anterior lo hacemos con la máscara de bits. De poder colocar el número verificamos si lo que tenemos hasta ahorita es mejor que el hecho de

calcular el subproblema de colocar i - 1 números mas el costo que genera colocar el número actual, de ser así conservamos lo que tenemos hasta ahorita, en caso contrario actualizamos ese valor y de esta forma provocaremos que al finalizar el algoritmo tengamos la solución en la posición $(N, 2^N - 1)$.
Con esto anterior podremos ver el costo mínimo que se generará al verificar por las diferentes permutaciones, pero tenemos el problema que no sabremos cuál de esas permutaciones es la que nos genera ese costo. Para remediar esto se propone una técnica parecida a la que se hace con el algoritmo de Dijkstra la cual consiste ir guardando la solución en una tabla muy parecida a la DP y al final reconstruirla de manera recursiva (para el presente caso la reconstrucción es iterativa). Esto es posible ya que cada vez que se actualiza la solución en la tabla de DP para una nueva casilla i sabemos explícitamente cuál es el número que se usó en esa casilla, por lo cual podemos ir actualizando la tabla que llamaremos de reconstrucción con el índice de este número cada vez que se encuentre una mejor solución.
Desde la línea 20 del algoritmo 2 podemos ver como se calcula la reconstrucción, la tabla *idx* se encarga de ir guardando esto anterior, la cual tiene la misma forma que la DP solamente que aquí almacenamos pares. El elemento izquierdo del par representa la máscara de bits y el elemento derecho representa el bit que se flipeó en esa máscara de bits, de esta forma hacemos la reconstrucción de abajo hacia arriba. Hay que tener en cuenta que al terminar la reconstrucción esta se encontrará al revés, por lo cual hay que voltear el arreglo y finalmente tendremos el resultado final.

3. Ejecuciones y resultados

A continuación se muestra una tabla con los resultados obtenidos de ejecutar 100 veces el programa, cada una con una instancia diferente. Se usaron las instancias proporcionadas en la tarea 2 y además se consiguieron 3 referentes a sudokus de tamaño 16x16, los resultados de dichas instancias se encuentran en las últimas tres filas de la tabla.
Es necesario mencionar que tras cada ejecución realizada por el programa se puso un tiempo de espera de 2 segundos para comenzar la siguiente ejecución, esto para permitir que la semilla aleatoria cambiara y obtener resultados no tan cercanos a los de la ejecución anterior, además, todas las inicializaciones de los sudokus fueron generadas aleatoriamente, es decir, no se utilizó una heurística constructiva.

Cuadro 1: Resultados de las 100 ejecuciones usando una generación aleatoria y la búsqueda local.

Instancia	Menor Fitness bl. sencilla	Menor Fitness bl. óptima	Tasa de éxito bl. sencilla	Tasa de éxito bl. óptima	Media Fitness bl. sencilla	Media Fitness bl. óptima	Media tiempo de ejecución bl. sencilla (seg)	Media tiempo de ejecución bl. óptima (seg)
Easy	2	0	0%	28%	3	22	0.0358684	0.0036767
David Filmer 1	6	2	0%	0%	12	7	0.0442347	0.0058913
David Filmer 2	4	2	0%	0%	12	7	0.0438622	0.0067342
Inkala	6	2	0%	0%	12	7	0.0428983	0.0079581
Medium	5	0	0%	5%	18	6	0.0416354	0.0048752
SD1	6	2	0%	0%	12	7	0.0441231	0.0061963
SD2	6	2	0%	0%	13	7	0.043254	0.0059216
SD3	4	3	0%	0%	13	7	0.0429436	0.0059273
Hard	5	0	0%	3%	31	6	0.0453188	0.0081781
16x16 1	24	8	0%	0%	70	17	1.2585	0.160018
16x16 2	26	10	0%	0%	52	21	1.24287	0.203881
16x16 3	27	8	0%	0%	59	18	1.27716	0.240503

De la tabla anterior, podemos notar varias diferencias con relación a la búsqueda local sencilla con relación a la óptima hecha con programación dinámica. La primera gran diferencia es que esta última fue capaz de resolver algunos sudokus de 3 instancias distintas (Easy, Medium, Hard) y la otra no tuvo resolución de ningún sudoku.
Otra diferencia apreciable es que la busqueda local optima genera en promedio mejores resultados que la busqueda sencilla, y esto anterior para todas las instancias probadas, se puede ver que en varias instancias los resultados fueron muy parecidos, esto ya que probablemente el nivel de dificultad de las instancias era el mismo.
Finalmente y no menos importante, podemos notar que la búsqueda local con DP tarda mucho menos en ejecutarse que la búsqueda sencilla. En la tabla se colocó la media del tiempo de ejecución en segundos y para todas las instancias la DP fue 10 veces más rápida que la otra búsqueda, incluso más en algunos casos. Esto anterior era de esperarse ya que la búsqueda sencilla es de tiempo $O(n!)$ y la búsqueda con DP es de tiempo $O(2^n n^2)$. Para instancias pequeñas esto no resulta en mucha diferencia pero para instancias grandes el cambio es muy notorio como se aprecia en las instancias de 16x16.

Notas y comentarios finales. Como conclusión se tiene que la implementación de la búsqueda local con DP resultó satisfactoria, así como la comparación realizada con la búsqueda local anterior mostró que esta versión es más óptima en varios sentidos, tales como el tiempo de ejecución y la calidad de las soluciones generadas.