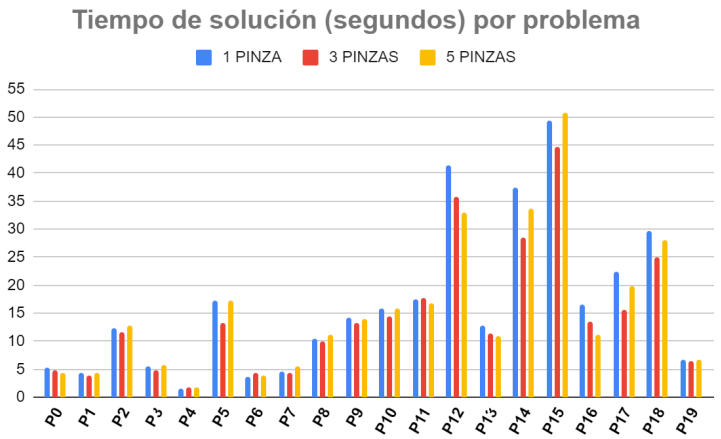
1. **Exactamente cómo debe ser modificado el archivo DCCubos.lp para que admita el uso de n brazos robóticos. Esto significa que en el mismo instante de tiempo, hasta n bloques podrían ser movidos a otro lugar.**

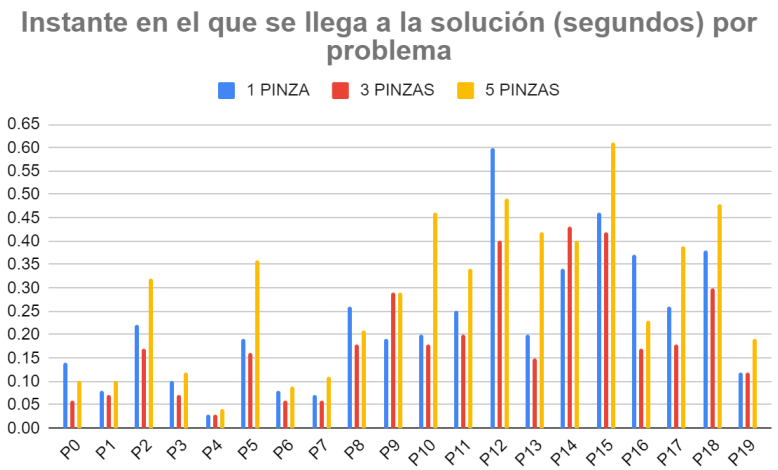
Para permitir el uso de *"n"* brazos robóticos, lo único que se debe hacer es modificar la restricción de cardinalidad que se muestra a continuación

```
 7
 8  %* ACA LAS PINZAS *%
 9  %* ACA LAS PINZAS *%
10  0{ejecutar(X,Y,T) : movimientoposible(X,Y,T)}1%* aca *% :- tiempo(T), T!=bound.
11  %* ACA LAS PINZAS *%
12  %* ACA LAS PINZAS *%
13
```

Como se puede observar, dejé resaltada la restricción de cardinalidad para facilitar las distintas iteraciones de 1, 3 y 5 pinzas para cada problema. Lo que indica la cardinalidad, es que se restringe la cantidad de movimientos posibles por unidad de tiempo entre 0 y 1, en el caso de una pinza. Para los distintos casos, tuve que cambiar el "1" por "3" o "5" respectivamente, para habilitar **hasta** 3 o 5 movimientos por turno.

2. **Medidas experimentales:** ***link a los datos***





3. **Conclusiones:**

En general, los resultados experimentales muestran que la solución propuesta funciona bien en una variedad de problemas con diferentes niveles de dificultad. En mi caso, implementé problemas con un rango entre 20 y 80 cubos, con distintas cantidades de torres y dificultades. Para el de tiempo de solución, se observa que el uso de tres brazos robóticos parece ser el óptimo, ya que ofrece una mejora significativa en la velocidad en comparación con una sola pinza y un tiempo similar al uso de cinco brazos robóticos.

En cuanto al instante en que se logra el objetivo, se observan resultados un poco disparejos, pero en general, 5 pinzas siempre parece ser más lento, seguido por 1 pinza, y luego 3 pinzas como la iteración más rápida.

Se podría concluir que el punto óptimo está en 3 pinzas ya que, con 1 pinza se restringe mucho el movimiento, y con 5 pinzas, el tiempo ahorrado por hacer más movimientos se ve superado por el tiempo pérdido considerando las muchas otras posibles soluciones.

4. **Minimize**

Para optimizar la cantidad de acciones en el programa usé:

```
#minimize{T : objetivo(T)}.
```

Para que se utilizara la menor cantidad de unidades de tiempo hasta llegar al estado objetivo.

En términos generales, al agregar la cláusula de minimize se mantuvieron tiempos o muy similares o drásticamente más bajos, como se observa en las columnas de tiempos de ejecución de los problemas 10 o 15 al aplicarles minimize. Esto se debe a que estamos encontrando la solución que requiere la menor cantidad de acciones para llegar al estado objetivo, lo que implica una reducción en tiempo de cómputo; sin embargo, me imagino que en problemas más simples (rápidos), el tiempo se mantiene similar debido a que el cómputo ahorrado al encontrar la solución más corta, se ve casi igualado por el cómputo gastado para demostrar que la solución presentada es efectivamente la más rápida. Es completamente opuesto en el caso de problemas más largos (problema 15), ya que el tiempo ahorrado supera considerablemente al tiempo de demostración, resultando en un neto negativo para el tiempo total de cómputo.