Este es un problema típico de búsqueda de una solución óptima mediante un algoritmo "vuelta atrás".

El problema puede resolverse satisfactoriamente con ayuda de una generalización que acepte los siguientes parámetros:

- La matriz de afinidades *as* entre clientes (parámetro de entrada) (la representación de dicha matriz incluye también el número total de clientes, tal y como se indica en la representación proporcionada en el código de apoyo)
- Un número de cliente a emparejar *c* (parámetro de entrada)
- La suma de afinidades de los clientes ya emparejados *afinidad_actual* (parámetro de entrada)
- Un vector de marcadores que proporcione información sobre qué clientes están ya emparejados. Este vector será un vector de booleanos *emparejados*, en el que el valor *emparejados*[*i*] de cada posición *i* indique si el cliente *i* está o no emparejado.
- La máxima afinidad conseguida hasta el momento *max_afinidad* (parámetro de entrada / salida)

Cada solución parcial al problema vendrá dada por: (i) la suma de las afinidades entre los miembros de las parejas ya emparejadas (representada por el parámetro *afinidad_actual*); y (ii) un número de cliente a emparejar (representado por el parámetro *c*).

Las soluciones parciales serán viables cuando cumplan las restricciones del problema: (i) no hay clientes emparejados consigo mismos; (ii) no hay clientes que hayan sido emparejados más de una vez; (iii) entre los clientes de cada pareja hay afinidades positivas.

La generación de las siguientes soluciones parciales viables dependerá de si el cliente *c* en sí mismo está o no está ya emparejado.

(i) Si *c* está ya emparejado, se considera como siguiente solución viable la determinada por *afinidad_actual* y *c+1*.

(ii) Si *c* no está emparejado, para cada cliente *c*' > *c* que no esté ya emparejado, y para el que *as*.afinidades[*c*][*c*'] > 0 y *as*.afinidades[*c'*][*c*] > 0, se genera una solución parcial dada por *afinidad_actual* + *as*.afinidades[*c*][*c*'] + *as*.afinidades[*c'*][*c*] y por *c*'+1.

Obsérvese que, en este proceso de generación, los marcadores *emparejados* juegan un papel fundamental para comprobar cuándo los clientes están o no emparejados.

Respecto al análisis de casos:

- El caso base se produce cuando el cliente a emparejar *c* coincide con el número total de clientes *as*.n_clientes. En este caso, si *afinidades_totales* excede a *max_afinidad*, se habrá encontrado una mejor solución, por lo que puede actualizarse *max_afinidad*.
- El caso recursivo se produce en cualquier otro caso: *c* es menor que *as*.n_clientes. En este caso, se procede a generar las siguientes soluciones parciales viables como se ha indicado anteriormente, realizándose, para cada solución generada, una llamada recursiva. En el caso de probar a emparejar *c* con *c*', antes de la llamada recursiva se actualiza el vector de marcadores. La marca se elimina cuando se vuelve de la llamada recursiva. Obsérvese, asimismo, que únicamente es necesario considerar los clientes cuyo número es mayor que *c*, ya que el emparejamiento de los que son menores que *c* se habrán tratado en otras ramificaciones del proceso recursivo. Con ello, únicamente

es necesario marcar el estado de emparejamiento de *c*', ya que el estado de emparejamiento de *c* no se consultará en ninguna parte del subproceso desencadenado por la llamada recursiva.

Este proceso puede realizarse mediante la generalización indicada. El algoritmo en sí se define por inmersión (i) inicializando todos los marcadores a *false*; y (ii) inicializando *max_afinidad* a 0 (con ello, no es necesario manejar explícitamente un booleano que indique si hay o no solución, ya que la ausencia de solución vendrá indicada por el valor 0 de *max_afinidad*).

Código resultante:

```
void encuentra_maxima_afinidad(const tMatrizAfinidad& as,
                                    unsigned int c,
                                    unsigned int afinidad_actual,
                                    bool emparejado[],
                                    unsigned int& max_afinidad) {
   if (c == as.n_clientes) {
     if (afinidad_actual > max_afinidad) {
          max_afinidad = afinidad_actual;
     }
    }
    else if (!emparejado[c]) {
        for (unsigned int p = c + 1; p < as.n_clientes; p++) {
           if (!emparejado[p]) {
               if (as.afinidades[c][p] > 0 && as.afinidades[p][c] > 0) {
                   emparejado[p] = true;
                   encuentra_maxima_afinidad(as, c + 1,
                                             afinidad_actual +
                                              as.afinidades[c][p] + as.afinidades[p][c],
                                             emparejado, max_afinidad);
                    emparejado[p] = false;
               }
            }
         }
       }
       else {
              encuentra_maxima_afinidad(as, c + 1,afinidad_actual,
                                        emparejado, max_afinidad);
       }
}

int maxima_afinidad(const tMatrizAfinidad& as) {
       bool emparejados[MAX_CLIENTES];
       unsigned int max_afinidad = 0;
       for (unsigned int c = 0; c < as.n_clientes; c++) {
              emparejados[c] = false;
       }
       encuentra_maxima_afinidad(as, 0, 0, emparejados,max_afinidad);
       return max_afinidad;
}
```

Obsérvese que si en el proceso de generación de soluciones parciales siguientes, si en lugar de poner

```
for (unsigned int p = c + 1; p < as.n_clientes; p++)
```

se pone

```
for (unsigned int p = 0; p < as.n_clientes; p++)
```

se generarán soluciones redundantes. Efectivamente, supongamos que hay un *c*' < *c* no emparejado. En la generación de la solución parcial actual debió tenerse en cuenta también el emparejamiento de *c*'. En dicho emparejamiento se debió ya producir, por tanto, el emparejamiento entre *c*' y *c*, por lo que volver a tenerlo en cuenta ahora introduce un nivel importante de redundancia. Este hecho puede comprobarse, por ejemplo, planteando el problema de calcular el número total de emparejamientos factibles. Se comprobará que con el

segundo bucle se están contando los mismos emparejamientos varias veces, con lo que el valor devuelto será significativamente más elevado que el real.

Una vez que tenemos configurado el algoritmo, podemos plantearnos la realización de posibles podas. Para ello, restringimos la viabilidad de las soluciones a que, además de todas las condiciones impuestas por el problema, *no pueda demostrarse que todas las soluciones completas a las que conducen no mejoran el mejor resultado obtenido hasta el momento*. En otro caso, seguir explorando las distintas posibilidades asociadas a la solución será estéril, ya que no se podrá descubrir ninguna mejora. Por tanto, la opción podrá *podarse* de manera segura.

Para hacer posible la poda podemos proporcionar a la generalización los siguientes argumentos extra:

(i) Una cota superior *cota_afin* de la afinidad máxima que puede conseguirse emparejando los clientes que restan por emparejar.

(ii) Un vector *max_afin_c* que indique, para cada cliente, la afinidad máxima que siente por los clientes de la agencia (este vector será necesario para actualizar el valor de *cota_afin* tras cada nuevo emparejamiento).

Con esta información, cuando se plantea emparejar a *c* con *c'*:

(1) Puede ajustarse la cota superior restando a *cota_afin* las afinidades máximas asociadas con *c* y *con c'* (estas afinidades pueden consultarse en el vector *max_afin_c*).

(2) En caso de que la afinidad de la nueva solución parcial planteada más la nueva cota así obtenida no supere el mejor resultado encontrado hasta el momento, se habrá demostrado que no es posible mejorar dicho resultado a partir de dicha solución parcial. Por tanto, la solución parcial podrá descartarse (podarse).

Código de la solución mejorada con poda de soluciones:

```
void encuentra_maxima_afinidad(const tMatrizAfinidad& as,
                               unsigned int c,
                               unsigned int afinidad_actual,
                               bool emparejado[],
                               unsigned int max_afin_c[],
                               unsigned int cota_afin,
                               unsigned int& max_afinidad) {
   if (c == as.n_clientes) {
      if (afinidad_actual > max_afinidad) {
          max_afinidad = afinidad_actual;
       }
    }
    else if (!emparejado[c]) {
      for (unsigned int p = c + 1; p < as.n_clientes; p++) {
         if (!emparejado[p]) {
            unsigned int ncota_afin = cota_afin - (max_afin_c[c] + max_afin_c[p]);
            unsigned int nafinidad = afinidad_actual +
                                     as.afinidades[c][p] + as.afinidades[p][c];
            if (as.afinidades[c][p] > 0 && as.afinidades[p][c] > 0 &&
                (ncota_afin + nafinidad) > max_afinidad) {
                emparejado[p] = true;
                encuentra_maxima_afinidad(as, c + 1,
                      nafinidad, emparejado,max_afin_c,ncota_afin, max_afinidad);
                emparejado[p] = false;
             }
          }
      }
    }
    else {
      encuentra_maxima_afinidad(as, c + 1,afinidad_actual, emparejado,
```

```
                                max_afin_c, cota_afin, max_afinidad);
        }
}


int maxima_afinidad(const tMatrizAfinidad& as) {
        bool emparejados[MAX_CLIENTES];
        unsigned int max_afin_c[MAX_CLIENTES];
        unsigned int cota_afin=0;
        for (unsigned int i = 0; i < as.n_clientes; i++) {
                max_afin_c[i] = 0;
                for (unsigned int j = 0; j < as.n_clientes; j++) {
                        if (as.afinidades[i][j] > max_afin_c[i]) {
                                max_afin_c[i] = as.afinidades[i][j];
                        }
                }
            cota_afin += max_afin_c[i];
        }
        unsigned int max_afinidad = 0;
        for (unsigned int c = 0; c < as.n_clientes; c++) {
                emparejados[c] = false;
        }
        encuentra_maxima_afinidad(as, 0, 0, emparejados,max_afin_c,
                                cota_afin, max_afinidad);
        return max_afinidad;
}
```

El bucle resaltado en `maxima_afinidad` determina las máximas afinidades profesadas por cada cliente, así como la cota superior inicial al resultado del problema.