

## Description of our algorithm

```
Main Algorithm {  
    SortEvents();  
  
    int ProblemHardness = ComputeProblemHardness();  
  
    bool HardProblem;  
  
    if (ProblemHardness > 825) {  
        HardProblem = true;  
    }  
    else {  
        HardProblem = false;  
    }  
  
    Set the domain for every event ;  
  
    int result = CBJ(0);  
  
    if (result == -1) {  
        Disinstantiate all events  
  
        FC(0);  
    }  
  
    ImproveSolution(Solution, HardProblem);  
}
```

## Backtracking algorithms

The following variables, constants, and functions are used:

- $N$  = Number of Events
- $K$  = Domain Size of an event ( $K$  varies depending on the current event)
- $current$  = the index of the current event
- $v$  is a one-dimensional array of size  $N$  that contains the current instantiations of the events
- $CBJ\_consistent(current)$  returns 1 if the current instantiation is consistent with past instantiations, and 0 otherwise
- $FC\_consistent(current)$  returns 1 if the current instantiation is consistent with future instantiations, and 0 otherwise
- $check(i, j)$  returns 1 if the consistency check between  $v[i]$  and  $v[j]$  succeeds, and 0 otherwise.
- $solution()$  stores the solution and computes its score.
- $merge(S1, S2)$  merges two sets:  $S1 = S1 \cup S2$
- $empty(S)$  empties a set
- $add(x, S)$  adds element  $x$  to set  $S$
- $max(S)$  returns the maximal element of set  $S$
- $restore(i)$  undoes the changes caused by the instantiation of an event
- $conf\_set[current]$  contains the past event which failed consistency checks with the current instantiation of the current event
- $checking[i][j]$  is set true if the current instantiation of event  $i$ , causes removal of some value from the domain of future event  $j$

## Conflict-Directed Backjumping (CBJ)

```
int CBJ_consistent(int current)
{
    int i ;

    for (i = 1 ; i < current ; i++) {
        if ( check(current, i) == 0) {
            add(i, conf_set[current]);
            return 0;
        }
    }
    return 1;
}

int CBJ(int current)
{
    int h, i, jump;

    every x computations, relax more soft constraint;

    if (current == N) {
        solution();
        return N;
    }

    empty(conf_set[current] );
    for (i = 0; i < K; i++) {
        v[current] = i;
        if (CBJ_consistent(current)) {
            jump = CBJ(current + 1);
            if (jump != current) {
                return jump;
            }
        }
    }

    h = max(conf_set[current]);
    merge(conf_set[h], conf_set[current]);
    return (h);
}
```

## Forward Checking (FC)

```
void restore(int i)
{
    int j, a;

    for (j = i + 1; j < N; j++) {
        if ( checking[i][j] ) {
            checking[i][j] = 0;

            for (a = 0; a < K; a++){
                if (domains[j][a] == i){
                    domains[j][a] = 0;
                }
            }
        }
    }
}

int FC_consistent (int current)
{
    int j, a;
    int old = 0, del = 0;

    for (j = current + 1; j < N; j++) {
        for (a = 0; a < K; a++) {
            if (domains[j][a] == 0) {
                old++;
                v[j] = a;
                if (check(current, j) == 0) {
                    domains[j][a] = current;
                    del++;
                }
            }
        }

        if (del) {
            checking[current][j] = 1;
        }
        if (old - del == 0) {
            return (j);
        }
    }

    return 0;
}
```

```
int FC (int current)
{
    int i, fail;

    if (current == N) {
        solution();
        return (N);
    }

    for (i = 0 ; i < K; i++) {
        if (domains[current][i]) {
            continue;
        }
        v[current] = i ;
        fail = FC_consistent (current) ;
        if (fail == 0) {
            FC(current + 1) ;
        }

        restore (current) ;
    }

    return (current-1) ;
}
```

## Algorithms to Improve the solution

```
void ImproveSolution(Solution mySol, bool HardProblem)
{
    if (HardProblem) {
        ILS(mySol, HardProblem);
    }
    else {
        SA_ILS_Combo(mySol);
    }
}
```

```
void SA_ILS_Combo (Solution mySol)
{
    LocalSearch(mySol);

    while (GetTime() < (MaxTime)) {
        SA(mySol);
        ILS(mySol, false);
    }
}
```

```
void SA(Solution* mySol)
{
    // Initialize these values
    double InitialT, T, InitialBeta, MaxBeta;
    int SolHistorySize;

    bool IsStuck = false;
    bool ReachedMax = false;

    // This prevent SA_OneMoveImproveSol from choosing end-of-day time slots
    DisAllowEndOfDays();

    Create SolHistory; // a list of solutions that we keep in memory

    int previousScore = mySol->Score;
    int count = 0;

    Solution * BestSol = new Solution(N);
    CopySolutions(BestSol, mySol);

    while (GetTime() < MaxTime) && ! IsStuck)
    {
        SA_OneMoveImproveSol(mySol, T);
        SA_TwoMoveImproveSol(mySol, T);
    }
}
```

```

if ( (mySol->Score) < (BestSol->Score)) {
    CopySolutions(BestSol, mySol);
    Beta = InitialBeta; // reset Beta
}

if (mySol->Score == previousScore)
{
    if (! (ApplyAcceptanceCondition(mySol, BestSol, SolHistory) ) )
    {
        CopySolutions(mySol, BestSol);
        T = T + Beta;
        Beta = Beta + Inc; //Inc is a constant

        if (Beta > MaxBeta) {
            if (ReachedMax) {
                IsStuck = true;

                //This allow SA_OneMoveImproveSol to
                // choose end-of-day time slots
                RelaxEndOfDays();
            }
            Beta = MaxBeta;
            ReachedMax = true;
        }
    }
    else {
        add mySol to SolHistory;
        T = T + InitialBeta;
    }
}

previousScore = mySol->Score;
}

CopySolutions(mySol, BestSol);
}

void SA_OneMoveImproveSol(Solution * mySol, double &T)
{
    for all events {
        previousScore = mySol->Score;

        try to allocate the event a new RoomTime such that the solution is
        feasible
    }
}

```

```

        if (! ApplyAcceptanceCondition(mySol->Score, previousScore, T) ) {
            Don't allocate this RoomTime and try another one;
        }

        if (we can't find any RoomTime for this event) {
            Allocate this event its initial RoomTime;
        }
    }
}

void SA_TwoMoveImproveSol(Solution * mySol, double &T)
{
    for all events {
        previousScore = mySol->Score;

        try to permutate the event with another event such that the solution is
        feasible

        if (! ApplyAcceptanceCondition(mySol->Score, previousScore, T) ) {
            Don't do this permutation and try another one;
        }
    }
}

void ILS(Solution * mySol, bool HardProblem)
{
    // That way, OneMoveImproveSol can't choose end-of-day time slots
    DisAllowEndOfDays();

    // The following values need to be initialized
    int PermutationNum , AllowedScoreIncrease, MaxPermutationNum;
    int MaxAllowedScoreIncrease;

    if (HardProblem) { // for a hard problem the max allowed score increase is higher
        MaxAllowedScoreIncrease is set to some value;
    }
    else { // for an easy problem this value is lower
        MaxAllowedScoreIncrease is set to some value;
    }
}

bool ReachedMax = false;
bool IsStuck = false;
int i;

```

```

int previousScore = 0;
int SolHistorySize = 5;
int SolHistoryCount = 0;

CreateList of Solutions;

// History of the most recent events pemutated by Pertubation
Create an N x N array History;

Solution* BestSol = new Solution(N);
BestSol->Score = 10000;

bool bOneMoveProgress = true;
bool bTwoThreeMoveProgress = true;
bool bstart = true;

LocalSearch(mySol);

BestSol->Score = mySol->Score;
CopySolutions(BestSol, mySol);

while ( (GetTime() < MaxTime) && !(HardProblem && IsStuck) )
{
    int k;// initialise to some value

    // If there is little time left we allow OneMoveImproveSol to select end of
    // day time slots
    if (GetTime() < (g_MaxTime - k))
    {
        // allows OneMoveImproveSol to choose end-of-day time slots
        RelaxEndOfDays();
    }

    Perturbation(mySol, PermutationNum, AllowedScoreIncrease, History);

    LocalSearch(mySol);

    if ( (mySol->Score) < (BestSol->Score) )
    {
        CopySolutions(BestSol, mySol);
        Reset AllowedScoreIncrease to its initial value ;
        Best_AllowedScoreIncrease=AllowedScoreIncrease;
        Reset PermutationNum to its initial value;
        Best_PermutationNum = PermutationNum;
    }
    else

```

```

    {
        if (! (ILS_ApplyAcceptanceCondition(mySol, BestSol, SolHistory,
        SolHistorySize, HardProblem)) )
        {
            CopySolutions(mySol, BestSol);
            Best_AllowedScoreIncrease += constant value;
            AllowedScoreIncrease= Best_AllowedScoreIncrease;
            Best_PermutationNum += constant value;
            PermutationNum = Best_PermutationNum ;
        }
        else {
            Insert mySol in SolHistory;

            Reset AllowedScoreIncrease to its initial value;
            Reset PermutationNum to its initial value;
        }
    }
    if (AllowedScoreIncrease > MaxAllowedScoreIncrease)
    {
        if (ReachedMax) {
            IsStuck = true;
            RelaxEndOfDays();
        }
        AllowedScoreIncrease = MaxAllowedScoreIncrease;
        ReachedMax = true;
    }
    if (PermutationNum > MaxPermutationNum)
    {
        PermutationNum = MaxPermutationNum;
    }
}
CopySolutions(mySol, BestSol, g_nEventNum);
}

```

```

bool ApplyAcceptanceCondition(Solution * mySol, Solution * BestSol, Solution **
History)

```

```

{
    If (mySol is too similar to BestSol)
        return false;

    for every solution in History
        if (mySol is too similar to that solution)
            return false;

    static double T = 3;
    double alpha = 0.75;

```

```

    double scoreDiff = (mySol->Score - BestSol->Score);

    double prob = exp(-(scoreDiff/T));
    T = alpha * T;

    return (true) with probability equal to (prob);
}

bool SA_ApplyAcceptanceCondition(int mySolScore, int BestSolScore, double &T)
{
    double alpha;
    Set alpha to some value;

    double scoreDiff = (double)(mySolScore - BestSolScore);

    double prob = exp(-(scoreDiff/T));

    T = alpha * T;

    return true with probability equal to (prob);
}

bool ILS_ApplyAcceptanceCondition(Solution * mySol, Solution * BestSol,
                                 Solution ** History, int HistorySize, bool Hard)
{
    if (mySol is too similar to BestSol)
        return false;

    if (mySol is too similar to any solution in History)
        return false;

    static double T = a constant;

    double alpha;

    if (Hard) { // for a hard problem alpha will be larger
        set alpha to some constant between 0 and 1;
    }
    else { // for an easy problem alpha will be smaller
        set alpha to some constant between 0 and 1;
    }

    double scoreDiff = (mySol->Score - BestSol->Score);

```

```

double prob = exp(-(scoreDiff/T));

T = alpha * T;

return true with probability equal to (prob);
}

void Perturbation(Solution* mySol, int PermutationNum,
                 int AllowedScoreIncrease, int ** History)
{
    int maxScore = mySol->Score + AllowedScoreIncrease;
    int PermutationDone = 0;

    while (PermutationDone < PermutationNum) {
        // according to History
        Select at random a pair of events that were not recently permuted ;

        if (the permutation of those events gives a feasible solution ) {
            if (this permutation does not increase the score higher than
                maxScore) {

                make this permutation;
                permutationDone++;
                UpdateHistory;
            }
        }

        if (there is no more permutations to be tried) {
            if (enough permutations were made) {
                break;
            }
            else if (not enough permutations were made and History is too full)
            {
                Reset History;
            }
            else {
                increase maxScore;
            }
        }
    }
}

```

```

void LocalSearch(Solution * mySol)
{
    int previousScore;
    bool bFirst = true;

    do {
        previousScore = mySol->Score;

        bOneMoveProgress = false;
        if (bfirst || bTwoThreeMoveProgress)
        {
            bOneMoveProgress = OneMoveImproveSol(mySol);
        }

        bTwoThreeMoveProgress = false;
        if (bfirst || bOneMoveProgress)
        {
            bTwoThreeMoveProgress = TwoThreeMoveImproveSol(mySol);
        }

        bfirst = false;
    } while( (mySol->Score < previousScore) && (GetTime() < MaxTime) );
}

// This algorithm tries to improve the score by allocating new RoomTimes to the events
// It returns true if it succeeds
bool OneMoveImproveSol(Solution * mySol)
{
    progress = false;
    success = false;

    do {
        for all events {
            try to allocate them a new RoomTime such that the solution is
            feasible and the score is lower;

            if (we succeed) {
                progress = true;
                success = true;
            }
        }
    } while (progress && (GetTime() < MaxTime));

    return success;
}

```

```

// This algorithms tries to improve the score by doing permutations between two or three
// events, it returns true if it succeeds
// A => B, B => A or A => B, B => C, C => A
bool TwoThreeMoveImproveSol(Solution * mySol)
{
    bool progress = false;
    bool success = false;

    do {
        for all events that have a penalty {
            try to find one or two events that we can permute with, such that
            the solution will be feasible and the score will be lower;

            if (we succeed) {
                progress = true;
                success = true;
            }
        }

    } while (progress && (GetTime() < MaxTime));

    return success;
}

```