# Decision Theoretic Troubleshooting

**Ariana Carnielli, Ivan Kachaikin**

**juin 05, 2020**

# Contents:

**class** DecisionTheoreticTroubleshooting.**TroubleShootingProblem**(*bayesian_network*, *costs*, *nodes_types*)

Classe créée pour représenter un problème de Troubleshooting. Contient des méthodes divers pour résoudre le problème. Utilise le module pyAgrum pour manipuler le réseau bayesien utilisé pour répresenter le problème. Les noeuds du réseau bayésien sont réferencés par des strings.

**Paramètres**

- **bayesian_network** (*pyAgrum.BayesNet*) – Représente le réseau bayésien (BN) modélisant un problème donné.
- **costs** (*list(dict)*) – Liste avec deux dictionnaires, le premier avec les coûts de réparation (exactes ou avec des minimun/maximun) et le deuxième avec les coûts d'observation des noeuds.
- **nodes_types** (*dict*) – Dictionnaire où les clés représent les noeuds du BN et les valeurs leurs types associés (set de string).

**bayesian_network**

Représente le réseau bayésien (BN) qui modélise un problème donné.

**Type** pyAgrum.BayesNet

**bay_lp**

Fait l'inference exacte pour le BN passé en argument.

**Type** pyAgrum.LazyPropagation

**costs_rep**

Dictionnaire de coûts où les clés représentent les noeuds du BN et les valeurs leurs coûts de reparation (float).

**Type** dict

**costs_rep_interval**

Dictionnaire de coûts où les clés représentent les noeuds du BN et les valeurs des listes avec les coûts minimum et maximum de reparation (floats).

**Type** dict

**costs_obs**

Dictionnaire de coûts où les clés représentent les noeuds du BN et les valeurs leurs coûts d'observation (float).

**Type** dict

**repairable_nodes**

Ensemble de noeuds qui correspondent aux éléments du système concerné qui peuvent être réparés.

**Type** set

**unrepairable_nodes**

Ensemble de noeuds qui correspondent aux éléments d'un système qui ne peuvent pas être réparés.

**Type** set

**problem_defining_node**

Noeud qui répresent le problème a être reglé (système fonctionnel où pas).

**Type** string

**observation_nodes**

Ensemble de noeuds qui correspondent aux éléments du système qui peuvent être observés.

**Type** set

**service_node**

Noeud qui répresent l'appel au service (appel à la réparation sûre du système).

**Type** string

**evidences**
Dictionnaire ou les clés répresentent les élements du système qui ont des evidences modifiés dans bay_lp (donc qui ont été réparés/observés) et les valeurs sont les inferences faites.

**Type** dict

**ECR_ECO_wrapper**(*debug=False*)
Calcule l'ECR myope pour chaque prochaine « observation-réparation » possible et l'ECO pour chaque prochaine observation globale possible.

**Paramètres** **debug** (*bool, facultatif*) – Si True, affiche des messages montrant le déroulement de l'algorithme.

**Renvoie**
— **chosen_node** (*string*) – Noeud choisi.
— **type_node** (*string*) – Type du noeud choisi (« repair » ou « obs »).
— **list_ecr** (*list(tuple)*) – ECRs des noeuds d'« observation-réparation ».
— **list_eco** (*list(tuple)*) – ECOs des noeuds d'observation globale.

**_compute_costs**(*costs*)
Prend en argument un dictionnaire de couts qui peut avoir des valeurs exactes ou des intervalles de valeurs (de la forme [minimum, maximum]) et le transforme en 2 dictionnaires, un avec les esperances de cout pour chaque clé et l'autre avec des intervalles de valeurs pour chaque clé.

**Paramètres** **costs** (*dict*) – Dictionnaire de couts où les clés représentent les noeuds du BN et les valeurs sont de nombres ou de listes de deux nombres.

**Renvoie**
— **expected_cost** (*dict*) – Dictionnaire où les clés représentent les noeuds du BN et les valeurs l'esperance de cout de ce noeud. Si la valeur initiale était déjà un nombre, ce nombre est seulement copié, sinon on considère que la valeur est une variable aléatoire avec une distribution uniforme dans l'intervalle et donc l'esperance est la moyenne des extremités de l'intervalle.
— **interval_cost** (*dict*) – Dictionnaire où les clés représentent les noeuds du BN et les valeurs sont des listes contenant les deux extremités des intervalles dans lequels les couts se trouvent. Si la valeur initiale était déjà un nombre, ce nombre est copié comme les deux extremités. Si la valeur initiale était un iterable, on le transforme en liste.

**_create_nodes**(*names*, *rep_string='_repair'*, *obs_string='_observation'*, *obs_rep_couples=False*)
Crée des noeuds de StrategyTree à partir de leurs noms dans le réseau Bayésien.

**Paramètres**
— **names** (*list(str)*) – Noms des noeuds de réparations/observations/ observations-réparations dans le réseau Bayésien à partir desquels on crée les noeuds.
— **rep_string** (*string, facultatif*) – Dans le cas où on ne considère pas des couples, on utilise ce paramètre comme un suffixe pour les noeuds de réparation pour les séparer de ceux d'observation.
— **obs_string** (*string, facultatif*) – Suffixe pour les noeuds d'observation.
— **obs_rep_couples** (*bool, facultatif*) – Variable boléenne qui indique si on suppose l'existance de couples « observation-réparation » dans l'arbre de stratégie.

**Renvoie** **nodes** – Liste de noeuds crées.

**Type renvoyé** list(*StrategyTree.NodeST*)

**_evaluate_all_st**(*feasible_nodes*, *obs_next_nodes=None*, *parent=None*, *fn_immutable=None*, *debug_nb_call=0*, *debug_iter=False*, *debug_st=False*, *obs_rep_couples=False*, *obs_obsolete=False*, *sock=None*)
Méthode récursive qui trouve le meilleur arbre de stratégie étant donné une configuration du problème en dénombrant tous les arbres admissibles.

**Paramètres**
— **feasible_nodes** (*list(StrategyTree.NodeST)*) – Lisste des noeuds admissibles qu'on a le droit d'utiliser pour construire l'arbre.

— **obs_next_nodes** (*list(list(str)), facultatif*) – Pile des attributs des arcs qui partent des noeuds déjà utilisés.
— **parent** (*list(tuple(StrategyTree.NodeST, StrategyTree.NodeST,* ) – StrategyTree.StrategyTree)), facultatif Pile des *parents* vers lesquels il faudra se retourner quand on remplit entièrement la branche courante.
— **fn_immutable** (*list(list(StrategyTree.NodeST)), facultatif*) – Pile des noeuds admissibles qu'on peut utiliser pour les branches différentes qui suivent un noeud d'observation.
— **debug_nb_call** (*int, facultatif*) – Profondeur de la récursivité.
— **debug_iter** (*bool, facultatif*) – Indique s'il faut afficher l'index de l'itération de l'appel.
— **debug_st** (*bool, facultatif*) – Précise s'il faut afficher tous les arbres intermédiaires.
— **obs_rep_couples** (*bool, facultatif*) – Indique si on suppose l'existance de couples « observation-réparation » dans l'arbre de stratégie soumis.
— **obs_obsolete** (*bool, facultatif*) – Indique si on suppose la possibilité des « observations obsolètes », i.e. qu'une observation devient obsolète en réparant une composante.
— **sock** (*socket.socket, facultatif*) – Socket de communication entre le processus qui effectue le calcul et celui qui met à jour l'interface. Paramètre nécessaire pour que le ProgressBar de l'interface marche proprement.

**_expected_cost_of_repair_internal**(*strategy_tree*, *evid_init=None*, *prob=1.0*, *obs_obsolete=False*)

Partie récursive de la fonction expected_cost_of_repair.

**Paramètres**
— **strategy_tree** (StrategyTree.StrategyTree) – Arbre de stratégie dont le coût il faut calculer.
— **evid_init** (*dict(str: str), facultatif*) – Dictionnaire d'évidences utilisé dans les appels récursifs.
— **prob** (*float, facultatif*) – Probabilité initiale.
— **obs_obsolete** (*bool, facultatif*) – Si True, on remet en cause les noeuds d'observation globale après une réparation.

**Renvoie** **ecr** – Coût espéré de réparation d'un arbre de stratégie fourni.

**Type renvoyé** float

**_next_node_id**()

Permet d'obtenir la prochaine valeur de *id* pour un noeud de StrategyTree à partir des ids qu'on utilise déjà stockés dans *self._nodes_ids_db_brute_force*.

**Renvoie** **next_id** – Prochain id qu'on peut utiliser avec garantie qu'il n'existe pas de noeuds déjà avec le même id dans l'arbre de stratégie courant.

**Type renvoyé** str

**_start_bay_lp**()

Ajoute des inférences vides aux noeuds du BN qui peuvent être modifiés (réparés/observés/appelés).Ces évidences ne changent pas les probabilités, elles servent pour qu'on puisse utiliser la méthode chgEvidence de pyAgrum à la suite.

**add_evidence**(*node*, *evidence*)

Fonction wrapper pour la fonction chgEvidence de l'objet bay_lp du type pyAgrum.LazyPropagation qui additionne une inference et mantient le dictionnaire evidences actualisé. L'evidence passé en argument ne doit pas être une evidence « vide » (des 1, utilisé plutôt la fonction remove_evidence).

**Paramètres**
— **node** (*string*) – Nom du noeud de bay_lp qui va être modifié.
— **evidence** – Nouvelle inference pour le noeud traité (généralement une string ici, cf. les types acceptés par chgEvidence)

**best_EVOI**()

Détermine la composante qui a la plus grande valeur espérée d'information (EVOI) correspondant à avoir plus d'information sur l'intervalle de valeur de son cout.

**Renvoie** Le nom du noeud de réparation avec la plus grande EVOI et la valeur d'EVOI correspondante.

**Type renvoyé** tuple(string, float)

**brute_force_solver**(*debug=False*, *mode='all'*, *obs_rep_couples=False*, *obs_obsolete=False*, *sock=None*)

Wrapper des différents algorithmes de recherche exhaustive qui calcule la solution exacte optimale étand donné un problème de Troubleshooting.

**Paramètres**

- **debug** (*bool / tuple, facultatif*) – Indique s'il faut afficher les résultats intermédiaires du calcul. Un tuple avec len(tuple) == 2 ou un booléean (équivalent au cas où on passe un tuple avec deux valeurs identiques). Le premier composant indique s'il faut afficher l'index de l'itération tandis que le deuxième précise s'il faut afficher tous les arbres intermédiaires.
- **mode** (*str, facultatif*) – Mode de calcul : soit "dp" pour la programmation dynamique, soit "all" pour le dénombrement complet.
- **obs_rep_couples** (*bool, facultatif*) – Indique si on suppose des couples « observation-réparation » dans l'arbre de stratégie.
- **obs_obsolete** (*bool, facultatif*) – Indique si on suppose la possibilité des « observations obsolètes », i.e. si c'est possible qu'une observation devient obsolète en réparant une composante.
- **sock** (*socket.socket, facultatif*) – Socket de communication entre le processus qui effectue le calcul et celui qui met à jour l'interface. Paramètre nécessaire pour que le ProgressBar de l'interface marche proprement.

**Renvoie**

- **best_st** (*StrategyTree.StrategyTree*) – Le meilleur arbre de stratégie trouvé.
- **best_ecr** (*float*) – Le coût espéré de réparation du meilleur arbre trouvé i.e. ECR(*best_st*).

**brute_force_solver_actions_only**(*debug=False*)

Cherche une séquence optimale de réparation par une recherche exhaustive en choisissant la séquence de meilleur ECR. Pour le cas où on ne considère que les actions de réparation il suffit de dénombrer toutes les permutations possibles d'un ensemble des actions admissibles.

**Paramètres** **debug** (*bool, facultatif*) – Si True, affiche des messages montrant le déroulement de l'algorithme.

**Renvoie**

- **min_seq** (*list(str)*) – Séquence optimale trouvée dont le coût est le plus petit possible.
- **min_ecr** (*float*) – Coût espéré de réparation correspondant à min_seq.

**brute_force_solver_tester**(*true_prices*, *epsilon*, *nb_min=100*, *nb_max=200*, *strategy_tree=None*, *mode='dp'*, *obs_rep_couples=False*, *true_prices_obs=None*)

Test empirique de la méthode brute_force_solver, ou bien de l'arbre de stratégie obtenu par l'algorithme. Le mécanisme selon lequel on teste la stratégie est exactement le même que celui utilisé au-dessus dans les autres testers.

**Paramètres**

- **true_prices** (*dict*) – Dictionnaire de prix de réparation des composantes réparables.
- **epsilon** (*float*) – Tolerance relative de la moyenne.
- **nb_min** (*int*) – Nombre minimum de répétitions à être realisées.
- **nb_max** (*int*) – Nombre maximum de répétitions à être realisées.
- **strategy_tree** (StrategyTree.StrategyTree, *facultatif*) – Arbre de stratégie qu'il faut tester; si rien passé, on calcule l'arbre avec la méthode brute_force_solver.

— **mode** (*string, facultatif*) – Paramètre à passer à la méthode brute_force_solver si on doit l'exécuter. Peut être égal à "all" pour le dénombrement complet ou "dp" pour la programmation dynamique.
— **obs_rep_couples** (*bool, facultatif*) – Paramètre à passer à la méthode brute_force_solver si on doit l'exécuter. Indique si on suppose l'existance de couples « observation-réparation » dans l'arbre de stratégie.
— **true_prices_obs** (*dict, facultatif*) – Dictionnaire de prix d'observations des composantes observables.

**Renvoie**
— **sortie_anti** (*bool*) – True en cas de sortie anticipée, False sinon.
— **costs** (*numpy.ndarray*) – Tableau avec les cout associés.
— **mean** (*float*) – Moyenne des couts.
— **std** (*float*) – Variance des couts.
— **cpt_repair** (*numpy.ndarray*) – Tableau avec le nombre de composantes réparées à chaque répétition.
— **cpt_obs** (*numpy.ndarray*) – Tableau avec le nombre d'observations globales faites à chaque répétition.

**compute_EVOIs**()
Calcule les valeurs espérées d'information (EVOIs) correspondant à avoir plus d'information sur l'intervalle de valeur des couts de réparation pour chaque composante réparable.

**Renvoie evoi** – Dictionnaire indexé par les noeuds réparables contenant la valeur d'une information plus précise du cout de réparation de ces noeuds.

**Type renvoyé** dict

**draw_true_prices**()
Tire au hasard des prix de réparation selon des lois uniformes sur les intervalles stockés dans self.costs_rep_interval.

**Renvoie** Dictionnaire avec prix de réparation.

**Type renvoyé** dict

**dynamic_programming_solver**(*feasible_nodes=None*, *evidence=None*, *debug_iter=False*, *debug_st=False*, *obs_rep_couples=False*, *prob=1.0*, *obs_obsolete=False*, *sock=None*, *debug_nb_call=0*)
Méthode récursive qui trouve le meilleur arbre de stratégie étant donné une configuration du problème en utilisant comme approche la programmation dynamique, supposant qu'un sous-arbre de l'arbre optimal est lui-même également optimal.

**Paramètres**
— **feasible_nodes** (*list(StrategyTree.NodeST), facultatif*) – Liste des noeuds admissibles qu'on a le droit d'utiliser pour construire l'arbre.
— **evidence** (*dict, facultatif*) – Dictionnaire des évidences initiales pour un appel de cette fonction.
— **debug_iter** (*bool, facultatif*) – Indique s'il faut afficher l""index de l'itération dans l'appel.
— **debug_st** (*bool, facultatif*) – Précise s'il faut afficher tous les arbres intermédiaires.
— **obs_rep_couples** (*bool, facultatif*) – Indique si on suppose l'existance de couples « observation-réparation » dans l'arbre de stratégie soumis.
— **prob** (*float, facultatif*) – Probabilité que le système ne marche toujours pas.
— **obs_obsolete** (*bool, facultatif*) – Indique si on suppose la possibilité des « observations obsolètes », i.e. qu'une observation devient obsolète en réparant une composante.
— **sock** (*socket.socket, facultatif*) – Socket de communication entre le processus qui effectue le calcul et celui qui met à jour l'interface. Paramètre nécessaire pour que le ProgressBar de l'interface marche proprement.

— **debug_nb_call** (*int, facultatif*) – Profondeur de la récursivité.

**elicitation**(*noeud*, *islower*)

Met à jour l'intervalle de valeurs de cout pour le noeud et son espérance en fonction de la réponse de l'utilisateur.

**Paramètres**

— **noeud** (*string*) – Nom du noeud à mettre à jour.
— **islower** (*bool*) – Représente la réponse à la question : Est-ce que le cout est plus petit que l'espérance courante ?

**elicitation_solver_tester**(*true_prices*, *epsilon*, *nb_min=100*, *nb_max=200*, *debug=False*)

Test empirique de la résolution avec élicitation. À chaque fois qu'on doit prendre une action, on vérifie d'abord s'il y a des questions à répondre et, si oui, on les répond toutes correctement selon true_prices. Ensuite, la méthode calcule la séquence d'actions itérativement à l'aide de myopic_solver et réalise au plus nb_max repetitions d'un système tiré au hasard, le tirage au hasard étant identique à celui de myopic_solver_tester. Si après nb_min repétitions l'erreur estimée est plus petite que epsilon, on fait une sortie anticipée. La fonction calcule les couts empiriques de réparation, en utilisant pour cela true_prices.

**Paramètres**

— **true_prices** (*dict*) – Dictionnaire de prix de réparation des composantes réparables.
— **epsilon** (*float*) – Tolerance relative de la moyenne.
— **nb_min** (*int*) – Nombre minimum de répétitions à être realisées.
— **nb_max** (*int*) – Nombre maximum de répétitions à être realisées.
— **debug** (*bool, facultatif*) – Si True, affiche des messages montrant le déroulement de l'algorithme.

**Renvoie**

— **sortie_anti** (*bool*) – True en cas de sortie anticipée, False sinon.
— **costs** (*numpy.ndarray*) – Tableau avec les cout associés.
— **mean** (*float*) – Moyenne des couts.
— **std** (*float*) – Variance des couts.
— **cpt_repair** (*numpy.ndarray*) – Tableau avec le nombre de composantes réparées à chaque répétition.
— **cpt_obs** (*numpy.ndarray*) – Tableau avec le nombre d'observations globales faites à chaque répétition.
— **cpt_questions** (*numpy.ndarray*) – Tableau avec le nombre de questions répondues à chaque répétition.

**expected_cost_of_repair**(*strategy_tree*, *obs_obsolete=False*)

Calcule le coût espéré de réparation étant donné un arbre de décision.

**Paramètres**

— **strategy_tree** (StrategyTree.StrategyTree) – Arbre de stratégie dont le coût il faut calculer.
— **obs_obsolete** (*bool, facultatif*) – Si True, on remet en cause les noeuds d'observation globale après une réparation.

**Renvoie** **ecr** – Coût espéré de réparation d'un arbre de stratégie fourni.

**Type renvoyé** float

**expected_cost_of_repair_seq_of_actions**(*seq*)

Calcule un coût espéré de réparation à partir d'une séquence d'actions donnée. On utilise la formule ECR = coût(C1 | E0) + P(C1 = Normal | E0) * coût(C2 | E1) + P(C1 = Normal | E0) * P(C2 = Normal | E1) * coût(C2 | E2) + ...

**Paramètres** **seq** (*list(str)*) – Séquence d'actions de réparations dont le coût espéré est à calculer.

**Renvoie** **ecr** – Coût espéré de réparation de la séquence donnée.

**Type renvoyé** float

**get_proba** (*node*, *value*)

Récupère à partir du réseau bayésien la probabilité que le noeud node ait la valeur value.

**Paramètres**

— **node** (*string*) – Nom du noeud de bay_lp dont on veut calculer la probabilité.

— **value** (*string*) – Valeur du noeud dont on veut calculer la probabilité.

**Renvoie** La probabilité P(node = value)

**Type renvoyé** float

**myopic_solver** (*debug=False*, *esp_obs=False*)

Implémente une étape du solveur myope. Étant donné l'état actuel du réseau, ce solveur utilise dans un premier temps le simple_solver_obs pour déterminer quelle action du type « observation-réparation » serait la meilleure. Ensuite, il calcule les coûts myopes espérés avec chaque observation possible et choisit à la fin la meilleure action à être prise.

Cette fonction est itérative et ne fait qu'un seul tour de l'algorithme myope car elle attend des nouvelles informations venues de l'utilisateur (résultat de l'observation si c'est le cas).

**Paramètres**

— **debug** (*bool, facultatif*) – Si True, affiche des messages montrant le déroulement de l'algorithme.

— **esp_obs** (*bool, facultatif*) – Si True, retourne en plus un dictionnaire indexé par les observations possibles et contenants leurs couts myopes espérés respectifs.

**Renvoie**

— **chosen_node** (*string*) – Le meilleur noeud de ce tour

— **type_node** (*string*) – Type du meilleur noeud (« repair » ou « obs »)

— **eco** (*dict*) – Retourné uniquement lorsque esp_obs vaut True. Dictionnaire des couts espérés des observations.

**myopic_solver_st** (*evid_init=None*)

Une méthode qui récupère un arbre de stratégie qu'on peut construire à partir de myopic_solver.

**Paramètres** **evid_init** (*dict, facultatif*) – Un dictionnaire des évidences par défaut.

**Renvoie** **strat_tree** – Un arbre de stratégie qu'on construit utilisant pas-à-pas une méthode myopic_solver en remplissant cet arbre en largeur.

**Type renvoyé** *StrategyTree.StrategyTree*

**myopic_solver_tester** (*true_prices*, *epsilon*, *nb_min=100*, *nb_max=200*, *debug=False*)

Test empirique de la méthode myopic_solver. Cette méthode calcule la séquence d'actions itérativement à l'aide de myopic_solver et réalise au plus nb_max repetitions d'un système tiré au hasard. À chaque observation globale, son résultat est tiré au hasard. Pour les paires « observation-réparation », on tire au hasard si la composante correspondante marche ou pas. Si oui, on ajoute juste le cout de l'observation et on continue, si non, on ajoute les couts d'observation et de réparation et on s'arrête (single fault assumption). Si on a une réparation simple sans observation associée, on ajoute directement le cout de réparation de la composante. Si après nb_min repétitions l'erreur estimée est plus petite que epsilon, on fait une sortie anticipée. La fonction calcule les couts empiriques de réparation, en utilisant pour cela true_prices.

**Paramètres**

— **true_prices** (*dict*) – Dictionnaire de prix de réparation des composantes réparables.

— **epsilon** (*float*) – Tolerance relative de la moyenne.

— **nb_min** (*int*) – Nombre minimum de répétitions à être realisées.

— **nb_max** (*int*) – Nombre maximum de répétitions à être realisées.

— **debug** (*bool, facultatif*) – Si True, affiche des messages montrant le déroulement de l'algorithme.

**Renvoie**

— **sortie_anti** (*bool*) – True en cas de sortie anticipée, False sinon.

— **costs** (*numpy.ndarray*) – Tableau avec les cout associés.

— **mean** (*float*) – Moyenne des couts.

— **std** (*float*) – Variance des couts.
— **cpt_repair** (*numpy.ndarray*) – Tableau avec le nombre de composantes réparées à chaque répétition.
— **cpt_obs** (*numpy.ndarray*) – Tableau avec le nombre d'observations globales faites à chaque répétition.

**myopic_wraper** (*debug=False*)
Interface textuelle pour le solveur myope. Utilise myopic_solver à chaque tour de boucle pour déterminer la meilleure action à prendre. Si c'est une observation, le résultat de l'observation est demandé, sinon on demande juste si l'action a résolu le problème. Les élicitations de couts ne sont pas implémentées. Les entrées de l'utilisateur ne sont pas sécurisées.

**Paramètres** **debug** (*bool, facultatif*) – Si True, affiche des messages montrant le déroulement de l'algorithme.

**noeud_ant** (*node*, *visites*)
Détermine tous les noeuds d'observation impactés par un changement du noeud node et qui sont antécesseurs de node, sans visiter les noeuds déjà dans l'ensemble des visites. Cette fonction est auxiliaire et n'a pas vocation à être appellée en dehors de la fonction principale observation_obsolete.

**Paramètres**
— **node** (*string*) – Nom du noeud don l'information a changé.
— **visites** (*set*) – Contient les noeuds déjà visités.

**Renvoie** **ant_obs** – Ensemble des noeuds d'observation affectés par node et qui sont antecesseurs de node sans être dans visites.

**Type renvoyé** set

**observation_obsolete** (*node*)
Étant donné un noeud dont l'information a changé, on détermine, à partir du réseau bayésien, tous les noeuds d'observation impactés par ce chagement.

**Paramètres** **node** (*string*) – Nom du noeud dont l'information a changé.

**Renvoie** **obs** – Ensemble contenant les noeuds d'observation impactés.

**Type renvoyé** set

**remove_evidence** (*node*)
Fonction wrapper pour la fonction chgEvidence de l'objet bay_lp du type pyAgrum.LazyPropagation qui retire une inference et mantient le dictionnaire evidences actualisé.

**Paramètres** **node** (*string*) – Nom du noeud de bay_lp qui va être modifié.

**reset_bay_lp** (*dict_inf={}*)
Reinitialise les inférences des noeuds du BN qui peuvent être modifiés (réparés/observés/appelés). Pour les noeuds dans dict_inf, l'inférence est mis à la valeur associé au noeud dans dict_inf, pour les autres l'inférence est mis à 1.

**Paramètres** **dict_inf** (*dict, facultatif*) – Dictionnaire où les clés sont des noeuds et les valeurs sont des inférences.

**simple_solver** (*debug=False*)
Solveur simple pour le problème du TroubleShooting. On ne prend pas en considèration des observations et on ne révise pas les probabilités, c'est-à-dire on ne met pas à jour les probabilités si on répare une composante. À cause de cela, ce solveur n'est pas iteractif et renvoie l'ordre de réparation entière (jusqu'au appel au service).

**Paramètres** **debug** (*bool, facultatif*) – Si True, affiche des messages montrant le déroulement de l'algorithme.

**Renvoie**
— **rep_seq** (*list*) – Séquence des noeuds à être réparés dans l'ordre.
— **exp_cost** (*float*) – Espérance du coût de réparation de cette séquence.

**simple_solver_obs**(*debug=False*)

Solveur simple pour le problème du Troubleshooting. On prend en considèration des paires « observation-réparation » (cf. définition dans l'état de l'art) mais pas les observations globales et on révise les probabilités, c'est-à-dire on met à jour les probabilités quand on « répare » une composante avant de calculer le prochaine composante de la séquence.

Le solveur n'est pas encore iteractif et renvoie l'ordre de réparation entière (jusqu'au appel au service). Cette choix à été fait car on utilise cet algorithme comme part de l'agorithme plus complexe et iteratif.

**Paramètres** **debug** (*bool, facultatif*) – Si True, affiche des messages montrant le déroulement de l'algorithme.

**Renvoie**

- **rep_seq** (*list*) – Séquence des noeuds à être réparés dans l'ordre.
- **exp_cost** (*float*) – Espérance du coût de réparation de cette séquence.

**simple_solver_obs_tester**(*true_prices*, *epsilon*, *nb_min=100*, *nb_max=200*)

Test empirique de la méthode simple_solver_obs. Cette méthode calcule la séquence d'actions à l'aide de simple_solver_obs et réalise au plus nb_max répétitions d'un système tiré au hasard : si on a une paire « observation-réparation », on tire au hasard si la composante correspondante marche ou pas. Si oui, on ajoute juste le cout de l'observation et on continue, si non, on ajoute les couts d'observation et de réparation et on s'arrête (single fault assumption). Si on a une réparation simple sans observation associée, on ajoute directement le cout de réparation de la composante. Si après nb_min repétitions l'erreur estimée est plus petite que epsilon, on fait une sortie anticipée. La fonction calcule les couts empiriques de réparation, en utilisant pour cela true_prices.

**Paramètres**

- **true_prices** (*dict*) – Dictionnaire de prix de réparation des composantes réparables.
- **epsilon** (*float*) – Tolerance relative de la moyenne.
- **nb_min** (*int*) – Nombre minimum de répétitions à être realisées.
- **nb_max** (*int*) – Nombre maximum de répétitions à être realisées.

**Renvoie**

- **sortie_anti** (*bool*) – True en cas de sortie anticipée, False sinon.
- **costs** (*numpy.ndarray*) – Tableau avec les cout associés.
- **mean** (*float*) – Moyenne des couts.
- **std** (*float*) – Variance des couts.
- **cpt_repair** (*numpy.ndarray*) – Tableau avec le nombre de composantes réparées à chaque répétition.

**simple_solver_tester**(*true_prices*, *epsilon*, *nb_min=100*, *nb_max=200*)

Test empirique de la méthode simple_solver. Cette méthode calcule la séquence d'actions à l'aide de simple_solver et réalise au plus nb_max repétitions d'un système tiré au hasard : à chaque fois qu'on a une probabilité qu'une action résoud le problème, on tire au hasard pour déterminer si le problème a effectivement été résolu ou pas suite à cette action. Si après nb_min repétitions l'erreur estimée est plus petite que epsilon, on fait une sortie anticipée. La fonction calcule aussi les couts empiriques de réparation, en utilisant pour cela true_prices. Cette méthode utilise la single fault assumption.

**Paramètres**

- **true_prices** (*dict*) – Dictionnaire de prix de réparation des composantes réparables.
- **epsilon** (*float*) – Tolerance relative de la moyenne.
- **nb_min** (*int*) – Nombre minimum de répétitions à être realisées.
- **nb_max** (*int*) – Nombre maximum de répétitions à être realisées.

**Renvoie**

- **sortie_anti** (*bool*) – True en cas de sortie anticipée, False sinon.
- **costs** (*numpy.ndarray*) – Tableau avec les cout associés.
- **mean** (*float*) – Moyenne des couts.
- **std** (*float*) – Variance des couts.
- **cpt_repair** (*numpy.ndarray*) – Tableau avec le nombre de composantes réparées à chaque répétition.

**class** DecisionTheoreticTroubleshooting.**bcolors**
Stockage de couleurs.

DecisionTheoreticTroubleshooting.**diff_dicts**(*left*, *right*)
Calcule la différence des dictionnaires *left* et *right* : les entrées de *left* dont la clé est aussi présente dans *right* sont supprimées, les autres sont gardées.

**Paramètres**
— **left** (*dict*) – Premier dictionnaire, duquel on supprime les clés apparaissant dans *right*.
— **right** (*dict*) – Deuxième dictionnaire, celui avec les clés qui doivent être supprimées de *left*.

**Renvoie** **res** – Résultat de la différence entre *left* et *right*.

**Type renvoyé** dict

DecisionTheoreticTroubleshooting.**merge_dicts**(*left*, *right*)
Fusionne deux dictionnaire passés sans les changer. Les couples (clé, valeur) du dictionnaire *right* sont plus prioritaires que celles de *left* ; c'est-à-dire, s'il existe une valeur associée à la même clé k dans les deux dictionnaires, on ajoute dans le résultat seulement celle qui appartient à *right*.

**Paramètres**
— **left** (*dict*) – Un des dictionnaires à fusionner, celui qui est moins prioritaire.
— **right** (*dict*) – L'autre dictionnaire à fusionner, celui qui est plus prioritaire.

**Renvoie** **res** – Résultat de la fusion.

**Type renvoyé** dict

DecisionTheoreticTroubleshooting.**shallow_copy_list_of_copyable**(*l*)
Crée une copie de profondeur 1 de la liste passée en argument : la liste est recopiée et remplie avec l'appel de la méthode copy() en chaque élément de la liste donnée.

**Paramètres** **l** (*list<Copyable>*) – La liste qui sera copiée. Chacun de ses éléments doit implémenter la méthode copy().

**Renvoie** **cl** – Copie de profondeur 1 de la liste passée en argument.

**Type renvoyé** list<Copyable>

DecisionTheoreticTroubleshooting.**shallow_copy_parent**(*parent*)
Crée une copie superficielle de *parent* (cf la méthode TroubleShootingProblem._evaluate_all_st ci-dessous).

**Paramètres** **parent** (*list(tuple(StrategyTree.NodeST, StrategyTree.NodeST, StrategyTree.StrategyTree))*) – Parent dont la copie il faut créer.

**Renvoie** **parent_copy** – Copie superficielle du parent passé.

**Type renvoyé** list(tuple(*StrategyTree.NodeST*, *StrategyTree.NodeST*, *StrategyTree.StrategyTree*))

**class** StrategyTree.**NodeST**(*id*, *cost*, *name=None*)
Classe qui représente un noeud plutôt abstrait d'un arbre de stratégie ; on remarque que cette classe ne dispose pas d'un enfant (il n'y a pas d'un attribut qui correspond à un noeud suivant), pourtant, on suppose que ses sous-classes en auront.

**Paramètres**
— **id** (*str*) – Identificateur unique d'un noeud.
— **cost** (*float*) – Un attribut qui correspond à « coût » d'un noeud.
— **name** (*str, facultatif*) – Un nom d'un noeud qui peut être pas unique ; si rien a été soumis, on pose que *_name* = *_id*.

**_id**
Identificateur unique d'un noeud.

**Type** str

**_cost**
Un attribut qui correspond à « coût » d'un noeud.

**Type** float

**_name**
Un nom d'un noeud qui peut être pas unique ; si rien a été passé, on pose que *_name* = *_id_*.

**Type** str

**__eq__**(*other*)
Overloading d'opérateur __eq__ ; on dit que deux noeuds sont égaux ssi ils ont les mêmes ids.

**Paramètres** **other** (NodeST) – Un autre noeud à comparer avec celui-ci.

**Renvoie** **comp_res** – True, si self._id == other._id ET si self et other ont le même type ; False, sinon.

**Type renvoyé** bool

**__str__**()
Overloading d'opérateur __str__.

**Renvoie** **corr_str** – Une représentation d'un noeud sous une forme de str.

**Type renvoyé** str

**add_child**(*child*)
Une méthode abstraite qui ajouterait un enfant d'un noeud.

**Paramètres** **child** (NodeST) – Un enfant à ajouter.

**bn_labels_children_association**()
Une méthode abstraite qui retournera un dictionnaire des associations entre des labels d'un réseau bayésien et des enfants d'un noeud.

**Renvoie** **da** – Un dictionnaire des associations concerné.

**Type renvoyé** dict

**copy**()
Une méthode qui retournerait une copie superficielle d'un noeud.

**Renvoie** **copy** – Une copie superficielle d'un noeud.

**Type renvoyé** *NodeST*

**get_child_by_attribute**(*attr*)
Une méthode abstraite qui retournerait un enfant d'un noeud correspondant à attr.

**Paramètres** **attr** (*str*) – Un attribut (un type) d'enfant qu'il faudrait retourner.

**Renvoie** **child** – Un enfant d'un noeud qui correspond à un attribut soumis.

**Type renvoyé** *NodeST*

**get_cost**()
Getter d'un attribut *_cost*.

**Renvoie** **_cost** – Une valeur de coût d'un noeud.

**Type renvoyé** float

**get_id**()
Getter d'un attribut *_id*.

**Renvoie** **_id** – Identificateur unique courant d'un noeud.

**Type renvoyé** str

**get_list_of_children**()
Une méthode abstraite qui permettrait d'obtenir une liste de tous les enfants d'un noeud.

**Renvoie** **list_of_children** – Une liste de tous les enfants d'un noeud.

**Type renvoyé** list(*NodeST*)

**get_name**()
Getter d'un attribut *_name*.

**Renvoie** **_name** – Un nom d'un noeud.

**Type renvoyé** str

**set_child_by_attribute**(*attr*, *child=None*)

Une méthode abstraite qui mettrait en place un enfant d'un noeud correspondant à attr.

**Paramètres**

— **attr** (*str*) – Un attribut (un type) d'enfant qu'il faudrait mettre en place.

— **child** (NodeST, *facultatif*) – Un enfant d'un noeud qui correspond à un attribut soumis et qu'il faut mettre en place.

**set_cost**(*cost*)

Setter d'un attribut *_cost*.

**Paramètres** **cost** (*float*) – Une valeur de coût d'un noeud à mettre en place.

**set_id**(*id*)

Setter d'un attribut *_id*.

**Paramètres** **id** (*str*) – Identificateur unique d'un noeud pour mettre en place.

**set_name**(*name*)

Setter d'un attribut *_name*.

**Paramètres** **name** (*str*) – Un nom d'un noeud pour mettre en place.

**class** StrategyTree.**Observation**(*id*, *cost*, *name=None*, *yes_child=None*, *no_child=None*, *obs_rep_couples=False*)

Classe qui représente un noeud de type correspondant à une action d'observation.

**Paramètres**

— **id** (*str*) – Identificateur unique d'un noeud.

— **cost** (*float*) – Un attribut qui correspond à « coût » d'un noeud.

— **name** (*str, facultatif*) – Un nom d'un noeud qui peut être pas unique ; si rien a été soumis, on pose que *_name* = *_id*.

— **yes_child** (NodeST, *facultatif*) – Un enfant d'un noeud qui correspond à une branche « yes ».

— **no_child** (NodeST, *facultatif*) – Un enfant d'un noeud qui correspond à une branche « no ».

— **obs_rep_couples** (*bool, facultatif*) – Un attribut sous la forme d'une variable boléenne qui indique si le noeud représente une couple observation-réparation ou pas.

**_id**

Identificateur unique d'un noeud.

**Type** str

**_cost**

Un attribut qui correspond à « coût » d'un noeud.

**Type** float

**_name**

Un nom d'un noeud qui peut être pas unique ; si rien a été passé, on pose que *_name* = *_id*.

**Type** str

**_yes_child**

Un enfant d'un noeud qui correspond à une branche « yes ».

**Type** *NodeST*

**_no_child**

Un enfant d'un noeud qui correspond à une branche « no ».

**Type** *NodeST*

**_obs_rep_couples**

Un attribut sous la forme d'une variable boléenne qui indique si le noeud représente une couple observation-réparation ou pas

**Type** bool

**add_child**(*child*)
Une méthode qui ajoute littéralement un enfant dans une liste des enfants d'un noeud. ATTENTION : ce méthode ne peut pas changer un enfant qui existe déjà ; pour cela, veuillez utiliser set_child !

**Paramètres** **child** (NodeST) – Un enfant à ajouter.

**bn_labels_children_association**()
Une méthode plutôt auxiliare qui retourne un dictionnaire des associations entre des labels d'un réseau bayésien et des enfants d'un noeud.

**Renvoie** **da** – Un dictionnaire des associations concerné.

**Type renvoyé** dict

**copy**()
Une méthode qui retourne une copie superficielle d'un noeud.

**Renvoie** **copy** – Une copie superficielle d'un noeud.

**Type renvoyé** *Observation*

**get_child_by_attribute**(*attr*)
Une méthode qui retourne un enfant d'un noeud correspondant à l'attribut de la branche soumise.

**Paramètres** **attr** (*str*) – Attribut de la branche.

**Renvoie** **child** – Un enfant correspondant à l'attribut.

**Type renvoyé** *NodeST*

**get_list_of_children**()
Une méthode qui retourne une liste avec tous les enfants d'un noeud.

**Renvoie** **list_of_children** – Une liste avec tous les enfants d'un noeud.

**Type renvoyé** list(*NodeST*)

**get_no_child**()
Getter d'un attribut *_no_child*.

**Renvoie** **_no_child** – Un enfant d'un noeud qui correspond à une branche « no ».

**Type renvoyé** *NodeST*

**get_obs_rep_couples**()
Getter d'un attribut *_obs_rep_couples*.

**Renvoie** **_obs_rep_couples** – Un attribut sous la forme d'une variable boléenne qui indique si le noeud représente une couple observation-réparation ou pas.

**Type renvoyé** bool

**get_yes_child**()
Getter d'un attribut *_yes_child*.

**Renvoie** **_yes_child** – Un enfant d'un noeud qui correspond à une branche « yes ».

**Type renvoyé** *NodeST*

**set_child_by_attribute**(*attr*, *child=None*)
Une méthode qui met en place un enfant d'un noeud correspondant à l'attribut de la branche soumise.

**Paramètres**
- **attr** (*str*) – Attribut de la branche.
- **child** (NodeST, *facultatif*) – Un enfant qu'il faut mettre en place.

**set_no_child**(*no_child=None*)
Setter d'un attribut *_no_child*.

**Paramètres** **no_child** (NodeST, *facultatif*) – Un enfant d'un noeud qui correspond à une branche « no ».

**set_obs_rep_couples**(*obs_rep_couples*)
Setter d'un attribut *_obs_rep_couples*.

**Paramètres** **obs_rep_couples** (*bool*) – Un attribut sous la forme d'une variable boléenne qui indique si le noeud représente une couple observation-réparation ou pas.

**set_yes_child**(*yes_child=None*)
Setter d'un attribut *_yes_child*.

**Paramètres** **yes_child** (NodeST, *facultatif*) – Un enfant d'un noeud qui correspond à une branche « yes ».

**class** StrategyTree.**Repair**(*id*, *cost*, *name=None*, *child=None*)
Classe qui représente un noeud de type plus précis : celui correspondant à une action de réparation.

**Paramètres**
- **id** (*str*) – Identificateur unique d'un noeud.
- **cost** (*float*) – Un attribut qui correspond à « coût » d'un noeud.
- **name** (*str, facultatif*) – Un nom d'un noeud qui peut être pas unique ; si rien a été soumis, on pose que *_name* = *_id*.
- **child** (NodeST, *facultatif*) – Un enfant d'un noeud.

**_id**
Identificateur unique d'un noeud.

**Type** str

**_cost**
Un attribut qui correspond à « coût » d'un noeud.

**Type** float

**_name**
Un nom d'un noeud qui peut être pas unique ; si rien a été passé, on pose que *_name* = *_id*.

**Type** str

**_child**
Un enfant d'un noeud, c'est-à-dire, un noeud suivant dans un arbre.

**Type** *NodeST*

**add_child**(*child*)
Une méthode qui ajoute littéralement un enfant dans une liste des enfants d'un noeud. ATTENTION : ce méthode ne peut pas changer un enfant qui existe déjà ; pour cela, veuillez utiliser set_child !

**Paramètres** **child** (NodeST) – Un enfant à ajouter.

**bn_labels_children_association**()
Une méthode plutôt auxiliare qui retourne un dictionnaire des associations entre des labels d'un réseau bayésien et des enfants d'un noeud.

**Renvoie** **da** – Un dictionnaire des associations concerné.

**Type renvoyé** dict

**copy**()
Une méthode qui retourne une copie superficielle d'un noeud.

**Renvoie** **copy** – Une copie superficielle d'un noeud.

**Type renvoyé** *Repair*

**get_child**()
Getter d'un attribut *_child*.

**Renvoie** **_child** – Un enfant d'un noeud.

**Type renvoyé** *NodeST*

**get_child_by_attribute**(*attr*)
Une méthode qui réalise une version abstraite de superclass ; comme ce type de noeud ne dispose que d'un seul enfant, on y retourne toujours cet enfant pour n'importe quel attr indiqué

**Paramètres** **attr** (*str*) – Un attribut (un type) d'enfant qu'il faudrait retourner.

**Renvoie** **child** – Un enfant d'un noeud.

**Type renvoyé** *NodeST*

**get_list_of_children**()

Une méthode qui retourne une liste qui contient tous les enfants d'un noeud ; pour ce cas-là alors, soit une liste avec un seul élément, soit une liste vide.

**Renvoie** **list_of_children** – Une liste de tous les enfants d'un noeud (soit une liste avec un seul élément, soit une liste vide).

**Type renvoyé** list(*NodeST*)

**set_child**(*child=None*)

Setter d'un attribut *_child*.

**Paramètres** **child** (NodeST, *facultatif*) – Un enfant d'un noeud.

**set_child_by_attribute**(*attr*, *child=None*)

Une méthode qui met en place un enfant d'un noeud correspondant à attr réalisant une méthode correspondante superclass.

**Paramètres**

- **attr** (*str*) – Un attribut (un type) d'enfant qu'il faut mettre en place.
- **child** (NodeST, *facultatif*) – Un enfant d'un noeud qui correspond à un attribut soumis et qu'il faut mettre en place.

**class** StrategyTree.**StrategyTree**(*root=None*, *nodes=None*)

Une classe pour représenter un arbre de stratégie qui fait face au problème de Troubleshooting.

**Paramètres**

- **root** (NodeST, *facultatif*) – Une racine de l'arbre, i.e. une action pour commencer.
- **nodes** (*list(*NodeST*)*, *facultatif*) – Une liste des noeuds de l'arbre.

**_root**

Une racine de l'arbre, i.e. une action pour commencer.

**Type** *NodeST*

**_nodes**

Une liste des noeuds de l'arbre.

**Type** list(*NodeST*)

**_adj_dict**

Un dictionnaire pour indiquer lesquels noeuds sont liés par des arcs.

**Type** dict

**fout_newline**

Un attribut utilisé pour indiquer le début d'une nouvelle ligne quand on transforme cet arbre ans un fichier textuel.

**Type** str

**fout_sep**

Un séparateur des attributs qu'on utilise quand on transforme cet arbre dans un fichier textuel.

**Type** str

**__str__**()

Une méthode qui réalise transformation de l'arbre vers str.

**Renvoie** **st_str** – Une représentation de l'arbre de stratégie en forme de str.

**Type renvoyé** str

**add_edge**(*parent*, *child*, *child_type=None*)

Une méthode qui permet d'ajouter un arc dans un arbre entre deux noeuds.

**Paramètres**

— **parent** (*str / NodeST*) – Un noeud qui doit être un parent dans cet arc, donc un noeud duquel cet arc part.
— **child** (*str / NodeST*) – Un noeud qui doit être un enfant dans cet arc, donc un noeud auquel cet arc entre.
— **child_type** (*str, facultatif*) – Un attribut de la branche du parent à laquelle il faut ajouter un enfant (par exemple si parent est une observation alors child_type est égal soit à "no", soit à "yes").

**add_node**(*node*)
Une méthode qui permet d'ajouter un.des nouveau.x noeud.s dans l'arbre.

**Paramètres** **node** (*NodeST / list(NodeST)*) – Noeud.s à ajouter.

**connect**(*root_with_subtree*, *root_child_type=None*)
Une méthode qui nous permet de connecter deux arbre, plus précisément, on connecte tout cet arbre avec celui soumis dans root_with_subtree en remplissant sa branche qui correspond à root_child_type.

**Paramètres**
— **root_with_subtree** (StrategyTree) – Un arbre vers la racine duquel on va connecter cet arbre.
— **root_child_type** (*str*) – Un attribut de la branche de la racine du root_with_subtree.

**Renvoie** **merged_tree** – Un arbre fusionné.

**Type renvoyé** *StrategyTree*

**copy**()
Une méthode qui retourne une copie superficielle de cet arbre.

**Renvoie** **copy** – Une copie superficielle de cet arbre-là.

**Type renvoyé** *StrategyTree*

**get_adj_dict**()
Un getter d'un attribut *_adj_dict*.

**Renvoie** **_adj_dict** – Une copie superficielle d'un dictionnaire d'adjacence.

**Type renvoyé** dict

**get_edges**()
Une méthode qui récupère tous les arcs de l'arbres.

**Renvoie** **edges** – Une liste de triplets où chaque celui edge correspond à un arc d'un graphe de manière que edge[0] est un parent, edge[1] est un enfant tandis que edge[2] est un attribut de la branche.

**Type renvoyé** list(tuple(*NodeST*, *NodeST*, str))

**get_node**(*id*)
Une méthode qui retourne un noeud exacte (en sens de l'objet dans mémoire vivant) de l'arbre avec id indiqué.

**Paramètres** **id** (*str / NodeST*) – Soit id de noeud, soit un noeud lui-même dont un clone (en sens d'id) il faut trouver dans l'arbre.

**Renvoie** **n** – Un noeud de l'arbre avec la même id que celui soumis.

**Type renvoyé** *NodeST*

**get_node_by_name**(*name*)
Une méthode qui retourne tous les noeuds de l'arbre dont les noms sont égaux à celui indiqué.

**Paramètres** **name** (*str*) – Un nom ou un noeud dont le nom on doit utiliser.

**Renvoie** **nodes** – Une liste de tous les noeuds de l'arbre qui ont le même nom que celui donné.

**Type renvoyé** list(*NodeST*)

**get_nodes**()
Un getter d'un attribut *_nodes*.

**Renvoie** **_nodes** – Une copie superficielle d'une liste des noeuds d'un arbre.

**Type renvoyé** list(*NodeST*)

**get_parent**(*child*)

Une méthode qui cherche et qui retourne un parent d'un noeud soumis dans l'arbre. Remarque : dans un arbre comme ça chaque noeud ne doit avoir qu'un seul parent.

**Paramètres** **child**(*str / NodeST*) – Un enfant dont le parent il faut trouver dans cet arbre-là.

**Renvoie** **parent** – Un parent trouvé d'un noeud soumis.

**Type renvoyé** *NodeST*

**get_root**()

Un getter d'un attribut *_root*.

**Renvoie** **_root** – Une racine de l'arbre.

**Type renvoyé** *NodeST*

**get_sub_tree**(*sub_root*)

Une méthode qui retourne un sous-arbre de cet arbre-là étant donné une sous-racine à partir de laquelle ce sous-arbre commence.

**Paramètres** **sub_root** (NodeST) – Une sous-racine de l'arbre qui est une racine de sous-arbre qu'il faut retourner.

**Renvoie** **sub_tree** – Un sous-arbre de cet arbre dont la racine est sub_root.

**Type renvoyé** *StrategyTree*

**remove_sub_tree**(*sub_root*)

Une méthode qui nous permet de supprimer un sous-arbre de cet arbre étant donné une sous-racine.

**Paramètres** **sub_root** (*str / NodeST*) – Une sous-racine d'un sous-arbre qu'il faut supprimer.

**Renvoie** **flag** – Une variable booléen qui est égale à True si on supprime quelque chose et False sinon.

**Type renvoyé** bool

**set_root**(*root*)

Un setter d'un attribut *_root*.

**Paramètres** **root** (NodeST) – Une racine à mettre en place.

**str_alt**()

Une méthode qui réalise une transformation particulière de l'arbre vers str.

**Renvoie** **st_str** – Une représentation de l'arbre de stratégie en forme de str.

**Type renvoyé** str

**str_alt_2**()

Une autre méthode qui réalise une transformation particulière de l'arbre vers str.

**Renvoie** **st_str** – Une représentation de l'arbre de stratégie en forme de str de manière alternative.

**Type renvoyé** str

**to_file**(*filemame='last_best_tree.txt'*)

Une méthode qui permet de sauvegarder cet arbre de stratégie sous en forme de fichier textuel. On y utilise le modèle suivant :

1) chaque noeud est représenté par une ligne dévouée : _id,_cost,_name,_type ; c'est bien possible de remplacer une virgule "," par un séparateur différent en précisant un attribut self.fout_sep de cette classe ; 2) chaque arc est représenté par une ligne dévouée : _id_parent,_id_child,_attribut ; où _attribut est le type d'arc pour cet arbre-là (par exemple "yes" ou "no" si parent est une Observation) ; 3) le fichier lui-même a la structure suivante : racine de l'arbre # ligne 1 [ligne vide] # ligne 2 noeud_1 # ligne 3 noeud_2 # ligne 4 … noeud_n # ligne n + 2 [ligne vide] # ligne n + 3 arc_1 # ligne n + 4 arc_2 # ligne n + 5 … arc_m # ligne n + m + 3

Cette méthode utilise également un attribut self.fout_newline pour représenter une signe qui indique le début d'une nouvelle ligne.

**Paramètres** **filemame** (*str, facultatif*) – Un nom de fichier où il faut sauvegarder l'arbre concerné.

**visualize**(*filename='last_best_strategy_tree.gv'*)

Une méthode qui nous permet d'afficher cet arbre de stratégie via un module graphviz. L'image construit est sauvegardé dans un fichier "filename.pdf".

**Paramètres** **filename** (*str, facultatif*) – Un nom de fichier où il faut sauvegarder une image obtenue.

StrategyTree.**st_from_file**(*filename='last_best_tree.txt'*, *sep=','*, *newline=None*)

Une méthode statique du module qui permet de créer un objet du type StrategyTree à partir du fichier qui suit un modèle fourni par la méthode StrategyTree.to_file.

**Paramètres**

- **filename** (*str, facultatif*) – Un nom du fichier où on stocke l'arbre sous en forme textuelle.
- **sep** (*str, facultatif*) – Un séparateur utilisé dans le fichier passé.
- **newline** (*str, facultatif*) – Une signe qui indique le début d'une nouvelle ligne.

**Renvoie** **stin** – Un arbre créé à partir des paramètres passés.

**Type renvoyé** *StrategyTree*

# Index des modules Python

## d



## s

# Index